GIÁO TRÌNH

CẤU TRÚC DỮ LIỆU

- 2003 -

Lời nói đầu

Cấu trúc dữ liệu là môn học chính yếu của chuyên ngành Công nghệ thông tin, là kiến thức nền tảng cho những người lập trình. Nhằm xây dựng một giáo trình vừa đảm bảo tính chuẩn mực của sách giáo khoa, vừa đáp ứng nhu cầu thực hành của chuyên viên. Chúng tôi đã tham khảo nhiều tài liệu giá trị của các tác giả trong và ngoài nước nhằm cung cấp kiến thức về môn học Cấu trúc dữ liệu một cách có hệ thống, nhiều vấn đề được minh hoạ trực quan và hướng dẫn theo từng bước lập trình cụ thể cho học viên.

Mặc dù rất nhiều cố gắng nhưng chắc chắn không thể tránh được thiếu sót. Chúng tôi rất mong nhận được các ý kiến đóng góp để giáo trình ngày càng được hoàn thiện.

Nhóm biên soạn

Chương 1: CÁC CẤU TRÚC DỮ LIỆU CƠ BẢN VÀ GIẢI THUẬT

1- Vai trò của cấu trúc dữ liệu:

Xây dựng một đề án tin học thực chất là chuyển bài toán thực tế thành một bài toán có thể giải quyết trên máy tính. Mà một bài toán thực tế bất kỳ đều bao gồm các đối tượng dữ liệu và các yêu cầu xử lý trên các đối tượng đó. Như vậy, để xây dựng một mô hình tin học phản ánh được bài toán thực tế cần chú trọng đến hai vấn đề:

- Tổ chức biểu diễn các đối tượng thực tế:

Các đối tượng dữ liệu thực tế rất đa dạng, phong phú và thường chứa đựng những quan hệ nào đó với nhau, do đó trong mô hình tin học của bài toán, cần phải tổ chức, xây dựng các cấu trúc thích hợp sao cho vừa có thể phản ánh chính xác các dữ liệu thực tế đó, vừa có thể dễ dàng dùng máy tính để xử lý. Công việc này được gọi là xây dựng *cấu trúc dữ liệu* cho bài toán.

- Xây dựng các thao tác xử lý dữ liệu:

Từ những yêu cầu xử lý thực tế, cần tìm ra các giải thuật tương ứng để xác định trình tự các thao tác máy tính phải tác động lên dữ liệu để cho ra kết quả mong muốn, đây là bước xây dựng *giải thuật* cho bài toán.

Trên thực tế khi giải quyết một bài toán trên máy tính chúng ta thường có khuynh hướng chỉ chú trọng đến việc xây dựng giải thuật mà quên đi tầm quan trọng của việc tổ chức dữ liệu trong bài toán. Cần nhớ rằng: Giải thuật phản ánh các phép xử lý, còn đối tượng xử lý của giải thuật lại là dữ liệu, chính dữ liệu chứa đựng các thông tin cần thiết để thực hiện giải thuật. Vì vậy để xác định được giải thuật phù hợp cần phải biết nó tác động đến loại dữ liệu nào (ví dụ để làm nhuyễn các hạt đậu, người ta dùng cách xay chứ không băm bằng dao, vì đậu sẽ văng ra ngoài và sẽ mất thời gian hơn nhiều) và khi chọn lựa cấu trúc dữ liệu cũng cần phải hiểu rõ những thao tác nào sẽ tác động đến nó (ví dụ để biểu diễn điểm số của sinh viên người ta dùng số thực thay vì chuỗi ký tự vì còn phải thực hiện thao tác tính trung bình từ những điểm số đó). Như vậy trong một đề án tin học, giải thuật và cấu trúc dữ liệu có mối quan hệ với nhau, chúng được thể hiện qua công thức nổi tiếng của nhà toán học người Thụy sĩ **Niklaus Wirth** - là tác giả của ngôn ngữ lập trình Pascal như sau:

Cấu trúc dữ liệu + Giải thuật = Chương trình

Với một cấu trúc dữ liệu đã chọn, sẽ có những giải thuật tương ứng, phù hợp. Khi cấu trúc dữ liệu thay đổi thường giải thuật cũng phải thay đổi theo để tránh việc xử lý gượng ép, thiếu tự nhiên trên một cấu trúc không phù hợp. Hơn nữa, một cấu trúc dữ liệu tốt sẽ giúp giải thuật xử lý trên đó có thể phát huy tác dụng tốt hơn, vừa đáp ứng nhanh vừa tiết kiệm tài nguyên, đồng thời giải thuật cũng dễ hiểu và đơn giản hơn.

Ví dụ: một chương trình quản lý điểm thi của sinh viên cần lưu trữ các điểm số của 4 sinh viên. Do mỗi sinh viên có 3 diểm số ứng với 3 môn học khác nhau nên dữ liệu có dạng bảng như sau:

Sinh viên	Môn 1	Môn 2	Môn 3
SV1	8	6	4
SV2	9	5	3
SV3	6	7	2
SV4	5	6	5

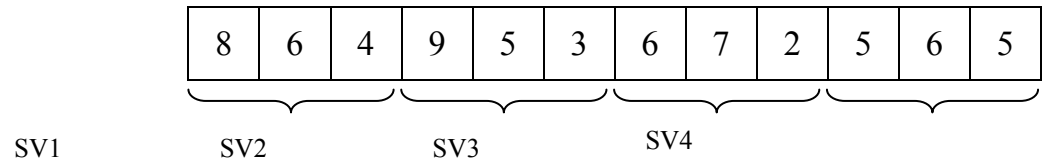
Chỉ xét thao tác xử lý là xuất điểm số các môn học của từng sinh viên. Giả sử có các phương án tổ chức lưu trữ sau:

Phương án 1: Sử dụng mảng một chiều

Có tất cả 4(sv) x 4(môn) = 12 điểm số cần lưu trữ, do đó khai báo mảng *diem* như sau:

```
int diem [12] = {  8   6   4
                   9   5   3
                   6   7   2
                   5   6   5 };
```

Khi đó trong mảng *diem* các phần tử sẽ được lưu trữ như sau:



Và truy xuất điểm số môn j của sinh viên i – (là phần tử tại dòng i, cột j trong bảng) - phải sử dụng một công thức xác định chỉ số tương ứng trong mảng *diem*:

bảngđiểm(dòng i, cột j) => diem[((i-1)*số cột) +j]

Ngược lại, với một phần tử bất kỳ trong mảng, muốn biết đó là điểm số của sinh viên nào, môn gì, phải dùng công thức xác định như sau:

diem[i] => diem[((i-1)*số cột) + j]

ở phương án này, thao tác xử lý được cài đặt như sau:

```
void indiem()
{  const  int  somon = 3;
   int sv, mon;
   for (int i=0; i<12; i++)
   {  sv = i/somon;
      mon=i % somon;
      printf("Điểm môn %d của SV %d là: %d", mon, sv,
               diem[i]);
   }
}
```

Phương án 2: Sử dụng mảng hai chiều

Khai báo mảng 2 chiều *diem* có kích thước 3 cột * 4 dòng như sau:

```
int diem [12] = {  {8   6   4},
                   {9   5   3},
                   {6   7   2},
                   {5   6   5}};
```

Khi đó trong mảng *diem* các phần tử sẽ được lưu trữ như sau:

	cột 0	**cột 1**	**cột 2**
Dòng 0	diem[0][0]=8	diem[0][1]=6	diem[0][2]=4
Dòng 1	diem[1][0]=9	diem[1][1]=5	diem[1][2]=3
Dòng 2	diem[2][0]=6	diem[2][1]=7	diem[2][2]=2
Dòng 3	diem[3][0]=5	diem[3][1]=6	diem[3][2]=5

Như vậy truy xuất điểm số môn j của sinh viên i là phần tử tại dòng i cột j trong bảng – cũng chính là phần tử nằm ở vị trí dòng i cột j trong mảng.

bảngđiểm(dòng i, cột j) => diem[i][j]

ở phương án này, thao tác xử lý được cài đặt như sau:

```
void indiem()
{  int  somon = 3, sosv = 4;
   for (int i=0; i<sosv; i++)
     for(int j=0; j<somon; j++)
        printf("Điểm môn %d của SV %d là: %d", j, i,
              diem[i][j]);
}
```

Nhận xét:

Ta có thể thấy rằng phương án 2 cung cấp một cấu trúc dữ liệu phù hợp với dữ liệu thực tế hơn phương án 1, do đó giải thuật xử lý trên cấu trúc dữ liệu của phương án 2 cũng đơn giản và tự nhiên hơn.

2- Các tiêu chuẩn đánh giá cấu trúc dữ liệu:

Qua phần trên ta đã thấy được vai trò và tầm quan trọng của việc lựa chọn một phương án tổ chức dữ liệu thích hợp trong một chương trình hay một đề án tin học. Một cấu trúc dữ liệu tốt phải thoả mãn các tiêu chuẩn sau:

Phản ảnh đúng thực tế: đây là tiêu chuẩn quan trọng nhất, quyết định tính đúng đắn của toàn bộ bài toán. Cần xem xét kỹ lưỡng cũng như dự trù các trạng thái biến đổi của dữ liệu trong chu trình sống để có thể chọn cấu trúc dữ liệu lưu trữ thể hiện chính xác đối tượng thực tế.

Ví dụ: một số trường hợp chọn cấu trúc dữ liệu sai:

- Chọn một số nguyên **int** để lưu trữ điểm trung bình của sinh viên (được tính theo công thức trung bình cộng của các môn học có hệ số), như vậy sẽ làm tròn mọi điểm số của sinh viên gây ra việc đánh giá sinh viên không chính xác qua điểm số. Trong trường hợp này phải sử dụng biến số thực để phản ảnh đúng kết quả của công thức tính thực tế cũng như phản ảnh chính xác kết quả học tập của sinh viên.
- Trong trường phổ thông, một lớp có 50 học sinh, mỗi tháng đóng quỹ lớp 1.000 đồng. Nếu chọn một biến số kiểu **unsigned int** (khả năng lưu trữ 0 – 65535) để lưu trữ tổng tiền qũy của lớp học trong tháng, nếu xảy ra trường hợp trong hai tháng liên tiếp không có chi hoặc tăng tiền đóng quỹ của mỗi học sinh lên 2.000 đồng thì tổng qũy lớp thu được là 100.000 đồng, vượt khỏi khả năng lưu trữ của biến đã chọn, gây nên tình trạng

tràn, sai lệnh. Như vậy khi chọn biến dữ liệu ta phải tính đến các trường hợp phát triển của đại lượng chứa trong biến để chọn liệu dữ liệu thích hợp. Trong trường hợp trên ta có thể chọn kiểu **long** (có kích thước 4 bytes, khả năng lưu trữ là -2147483648 → 2147483647) để lưu trữ tổng tiền quỹ lớp.

Phù hợp với các thao tác xử lý: tiêu chuẩn này giúp tăng tính hiệu quả của đề án: phát triển các thuật toán đơn giản, tự nhiên hơn; chương trình đạt hiệu quả cao hơn về tốc độ xử lý.

Ví dụ: một số trường hợp chọn cấu trúc dữ liệu không phù hợp

- Khi cần xây dựng một chương trình soạn thảo văn bản, các thao tác xử lý thường xảy ra là chèn, xoá, sửa các ký tự trên văn bản. Trong thời gian xử lý văn bản, nếu chọn cấu trúc lưu trữ văn bản trực tiếp lên tập tin thì sẽ gây khó khăn khi xây dựng các giải thuật cập nhật văn bản và làm chậm tốc độ xử lý của chương trình vì phải làm việc trên bộ nhớ ngoài. Trường hợp này nên tìm một cấu trúc dữ liệu có thể tổ chức có thể tổ chức ở bộ nhớ trong để lưu trữ văn bản suốt thời gian soạn thảo.

Tiết kiệm tài nguyên hệ thống: cấu trúc dữ liệu chỉ nên sử dụng tài nguyên hệ thống vừa đủ để đảm nhiệm được chức năng của nó. Thông thường có hai loại tài nguyên cần lưu tâm nhất là CPU và bộ nhớ. Tiêu chuẩn này nên cân nhắc tuỳ vào tình huống cụ thể khi thực hiện đề án. Nếu tổ chức sử dụng đề án cần có những xử lý nhanh thì chọn cấu trúc dữ liệu có yếu tố tiết kiệm thời gian xử lý ưu tiên hơn tiêu chuẩn sử dụng tối ưu bộ nhớ, và ngược lại.

Ví dụ: một số trường hợp chọn cấu trúc dữ liệu gây lãng phí

- Sử dụng biến **int** (2 bytes) để lưu trữ một giá trị thông tin về ngày trong tháng. Vì một tháng chỉ có thể nhận các giá trị từ 1-31 nên chỉ cần sử dụng biến char (1 byte) là đủ.
- Để lưu trữ danh sách nhân viên trong công ty mà sử dụng mảng 1000 phần tử. Nếu số lượng nhân viên thật sự ít hơn 1000 (bị giảm hoặc biên chế không đủ) thì gây lãng phí. Trường hợp này cần có một cấu trúc dữ liệu linh động hơn mảng – ví dụ danh sách liên kết.

3- Khái niệm về kiểu dữ liệu:

Máy tính chỉ có thể lưu trữ dữ liệu ở dạng nhị phân sơ cấp. Để phản ảnh được dữ liệu thực tế đa dạng và phong phú, cần phải xây dựng các phép ánh xạ, những quy tắc tổ chức phức tạp che lên tầng dữ liệu thô, nhằm đưa ra những khái niệm logic về hình thức lưu trữ khác nhau thường được gọi là kiểu dữ liệu. Như đã phân tích ở mục trước, giữa hình thức lưu trữ dữ liệu và các thao tác xử lý trên đó có quan hệ mật thiết với nhau. từ đó có thể đưa ra một định nghĩa cho kiểu dữ liệu như sau:

3.1- Định nghĩa kiểu dữ liệu

Kiểu dữ liệu T được xác định bởi một bộ **<V,O>**, với:

V: tập các giá trị hợp lệ mà một đối tượng kiểu T có thể lưu trữ.

O: tập các thao tác xử lý có thể thi hành trên đối tượng kiểu T.

Ví dụ: – Giả sử có kiểu dữ liệu **mẫu tự** = $<V_c,O_c>$ với

V_c = {a-z, A-Z}

O_c = {lấy mã ASCII của ký tự, biến đổi ký tự thường thành ký tự hoa…}

– Giả sử có kiểu dữ liệu **số nguyên** = $<V_i,O_i>$ với

V_i = {–32768 .. 32767}

O_i = {+, – , *, /, %}

Như vậy, muốn sử dụng một kiểu dữ liệu cần nắm vững cả nội dung dữ liệu được phép lưu trữ và các xử lý tác động trên đó.

Các thuộc tính của một kiểu dữ liệu bao gồm:

- Tên kiểu dữ liệu
- Miền giá trị
- Kích thước lưu trữ
- Tập các toán tử tác động lên kiểu dữ liệu

3.2- Các kiểu dữ liệu cơ bản

Các loại dữ liệu cơ bản thường là các loại dữ liệu đơn giản, không có cấu trúc. Chúng thường là các giá trị vô hướng như các số nguyên, số thực, các ký tự, các giá trị logic, … Các loại dữ liệu này do tính thông dụng và đơn giản của nó, thường được các ngôn ngữ lập trình cấp cao xây dựng sẵn như một thành phần của ngôn ngữ để giảm nhẹ công việc cho người lập trình. Chính vì vậy đôi khi người ta còn gọi chúng là các kiểu dữ liệu định sẵn.

Thông thường, các kiểu dữ liệu cơ bản gồm:

- Kiểu có thứ tự rời rạc: số nguyên, ký tự, logic, liệt kê, miền con, …
- Kiểu không rời rạc: số thực

Tùy ngôn ngữ lập trình, các kiểu dữ liệu định nghĩa sẵn có thể khác nhau đôi chút. Với ngôn ngữ C, các kiểu dữ liệu này chỉ gồm số nguyên, số thực, ký tự. Và theo quan điểm của C, kiểu ký tự thực chất cũng là kiểu số nguyên về mặt lưu trữ, chỉ khác về cách sử dụng. Ngoài ra, giá trị logic ĐÚNG (TRUE) và giá trịc logic SAI (FALSE) được biểu diễn trong C như là các giá trị nguyên khác zero và zero. Trong khi đó PASCAL định nghĩa tất cả các kiểu dữ liệu cơ sở đã liệt kê ở trên và phân biệt chúng một cách chặt chẽ. Trong giáo trình này sử dụng ngôn ngữ C để minh hoạ.

Các kiểu dữ liệu định sẵn trong C gồm:

Tên kiểu	Kthước	Miền giá trị	Ghi chú
char	1 byte	-128...127	Có thể dùng như số nguyên 1 byte có dấu hoặc kiểu ký tự
unsign char	1 byte	0...255	số nguyên 1 byte không dấu
int	2 bytes	-32768...32767	
unsigned int	2 byte	0..65355	gọi tắt là unsign
long	4 bytes	-2^{32} ... $2^{31}-1$	
unsigned long	4 bytes	0 ... $2^{32}-1$	
float	4 bytes	3.4E-38 …3.4E38	giới hạn chỉ trị tuyệt đối. Các giá trị <3.4E38 được coi bằng 0. Tuy nhiên kiểu float chỉ có 7 chữ số có nghĩa
double	8 bytes	1.7E-3.8 …1.7E3.8	
long double	10 bytes	3.4E-4932 …1.1E4932	

Một số lưu ý:

- Kiểu char trong C có thể dùng theo hai cách: số nguyên một byte hoặc ký tự.
- Trong C không định nghĩa kiểu logic (boolean) mà xem một số nguyên khác không là TRUE và giá trị 0 là FALSE khi cần xét các giá trị logic.
- Như vậy trong C thực chất chỉ có 2 loại dữ liệu cơ bản là số nguyên và số thực.

- Trong C các số nguyên có thể được thể hiện trong 3 hệ cơ số là hệ thập lục phân, hệ thập phân và hệ bát phân.

3.3- Các kiểu dữ liệu có cấu trúc

Các kiểu dữ liệu cơ sở rất thô sơ, đơn giản và không phản ánh được các đối tượng tự nhiên cũng như đầy đủ yếu tố của sự vật thực tế, do đó dẫn đến nhu cầu phải xây dựng các kiểu dữ liệu mới dựa trên việc tổ chức, liên kết các thành phần dữ liệu có kiểu dữ liệu cơ sở đã được định nghĩa. Những kiểu dữ liệu được xây dựng như thế gọi là kiểu dữ liệu có cấu trúc. Hầu hết các ngôn ngữ lập trình đều cài đặt sẵn một số kiểu có cấu trúc cơ bản như mảng chuỗi, tập tin, bản ghi (struct trong C hay record trong Pascal),… và cung cấp cơ chế cho lập trình viên tự định nghĩa kiểu dữ liệu mới.

Ví dụ: Để mô tả một đối tượng nhân viên, cần quan tâm đến các thông tin sau:
- Mã nhân viên : chuỗi ký tự
- Tên nhân viên: chuỗi ký tự
- Ngày sinh : kiểu ngày tháng năm
- Nơi sinh : chuỗi ký tự
- Lương căn bản: số nguyên

để mô tả thông tin điểm thi ta dùng kiểu dữ liệu cơ sở:

```
long    luongcb;
```

Các thông tin khác đòi hỏi phải dùng kiểu có cấu trúc sau:

```
char    manv[10];
char    tennv[30];
char    noisinh[50];
```

Để thể hiện thông tin về ngày tháng năm sinh cần phải xây dựng một kiểu bản ghi:

```
typedef  struct  tagDate{
        char    ngay;
        char    thang;
        char    nam;
} date;
```

Từ trên, ta có thể xây dựng kiểu dữ liệu thể hiện thông tin về một nhân viên:

```
typedef  struct  tagNhanVien{
        char    manv[10];
        char    tennv[30];
        date    ngaysinh;
        char    noisinh[50]
        long    luongcb;
}
```

Giả sử đã có cấu trúc phù hợp để lưu trữ một nhân viên, nhưng thực tế lại cần quản lý nhiều nhân viên, lúc đó nảy sinh nhu cầu xây dựng kiểu dữ liệu mới. Mục tiêu của việc nghiên cứu cấu trúc dữ liệu chính là tìm những phương cách thích hợp để tổ chức, liên kết dữ liệu, hình thành các kiểu dữ liệu có cấu trúc từ những kiểu dữ liệu đã được định nghĩa.

3.4- Một số kiểu dữ liệu có cấu trúc cơ bản

a. Kiểu chuỗi ký tự:

Chuỗi ký tự là một trong các kiểu dữ liệu có cấu trúc đơn giản nhất và thường các ngôn ngữ lập trình đều định nghĩa nó như một kiểu cơ bản. Do tính thông dụng của kiểu chuỗi ký tự các ngôn ngữ lập trình luôn cung cấp sẵn một bộ các hàm thư viên xử lý trên kiểu dữ liệu này. Trong C, thư viện các hàm xử lý chuỗi ký tự rất đa dạng và phong phú. Các hàm này được đặt trong thư viện string.lib của C.

Chuỗi ký tự trong C được cấu trúc như một chuỗi liên tiếp các ký tự kết thúc bằng ký tự có mã ASCII bằng 0 (NULL character). Như vậy, giới hạn chiều dài của một chuỗi ký tự trong C là 1 segment (chứa tối đa 65535) ký tự, ký tự đầu tiên được đánh số thứ tự là ký tự thứ 0.

Ta có thể khai báo một chuỗi ký tự theo một số cách sau đây:

```
char   s[10];     // khai báo một chuỗi ký tự có chiều dài
                  // tối đa là 10 (kể cả ký tự kết thúc)
char  s[] = "ABC"; // khai báo một chuỗi ký tự s có
                  //chiều dài bằng chiều dài của chuỗi "ABC"
                  //và giá trị khởi đầu của S là "ABC"
char   *s = "ABC"; // giống cách khai báo trên
```

Ở ví dụ trên ta thấy rằng một hằng chuỗi ký tự được thể hiện bằng một chuỗi ký tự đặt trong cặp dấu nháy kép.

Các thao tác trên chuỗi ký tự rất đa dạng. Sau đây là một số thao tác thông dụng:

- So sánh 2 chuỗi strcmp
- Sao chép 2 chuỗi strcpy
- Kiểm tra 1 chuỗi nằm trong chuỗi kia strstr
- Cắt một từ ra khỏi một chuỗi strok
- Đổi 1 số ra chuỗi itoa
- Đổi 1 chuỗi ra số atoi, atof
- Đổi 1 hay một số giá trị ra chuỗi sprintf
- Nhập một chuỗi gets
- Xuất một chuỗi puts

b. Kiểu mảng (array):

Mảng là kiểu dữ liệu trong đó mỗi phần tử của nó là một tập hợp có thứ tự các giá trị có cùng cấu trúc, thường được lưu trữ liên tiếp nhau trong bộ nhớ. Mảng có thể một chiều hay nhiều chiều. Một dãy số chính là hình tượng của mảng 1 chiều, ma trận là hình tượng của mảng 2 chiều.

Một điều đáng lưu ý là mảng 2 chiều có thể coi là mảng một chiều trong đó mỗi phần tử của nó là 1 mảng một chiều. Tương tự như vậy, một mảng n chiều có thể coi là mảng 1 chiều trong đó mỗi phần tử là 1 mảng n-1 chiều.

Trong C mảng 1 chiều được khai báo như sau:

<Kiểu dữ liệu><Tên biến>[<Số phần tử>];

Ví dụ để khai báo một mảng a chứa tối đa 50 số nguyên ta khai báo như sau:

```
int     a[50];
```

Ta có thể vừa khai báo vừa gán giá trị khởi động cho một mảng như sau:

```
int     a[5] = {3,5,7,-9,16};
```

Trong trường hợp trên C cho phép ta khai báo một cách tiện lợi hơn như sau:

```
int     a[] = {3,5,7,-9,16};
```

Như trên ta thấy không cần chỉ ra số lượng phần tử cụ thể trong khai báo. Trình biên dịch sẽ tự động là việc này cho chúng ta.

Tương tự ta có thể khai báo một mảng 2 chiều hay nhiều chiều theo cú pháp sau:

<Kiểu dữ liệu><Tên biến>[<Số phần tử1>][<Số phần tử2>];

Ví dụ ta có thể khai báo:

```
int     a[50][40];
```

hay

```
int     a[][] = { {1,5,6,-7,4},
                  {-8,9,-2,5,3},
                  {4,-6,7,3,1} }
```

(mảng a sẽ có kích thước là 3x5)

c. Kiểu cấu trúc (struct):

Kiểu cấu trúc là kiểu dữ liệu mà trong đó mỗi phần tử của nó là tập hợp các giá trị có thể khác cấu trúc. Kiểu mẫu tin cho phép chúng ta mô tả các đối tượng có cấu trúc phức tạp.

- Khai báo tổng quát của kiểu struct như sau:

```
typedef  struct <tên kiểu struct> {
        <kiểu dl> <tên trường>;
        <kiểu dl> <tên trường>;
        …
}[<Name>];
```

- Ví dụ để khai báo thông tin về một sinh viên ta có thể khai báo một kiểu dữ liệu như sau:

```
struct  tagNhanVien
{       char    MaSo[5];
        char    HoTen[30];
        int     NamSinh;
        char    GioiTinh;
        char    DiaChi[50];
}
```

Kiểu cấu trúc bổ sung nhiều thiếu sót của kiểu mảng, giúp ta có khả năng thể hiện các đối tượng đa dạng của thế giới hiện thực vào trong máy tính một cách dễ dàng, chính xác hơn.

d. Kiểu Union:

Kiểu union là một dạng cấu trúc dữ liệu đặc biệt của ngôn ngữ C. Nó rất giống với kiểu cấu trúc. Chỉ khác một điều, trong kiểu union, các trường được phép dùng chung một vùng nhớ, ta có thể truy xuất dưới các dạng khác nhau.

Khai báo tổng quát của kiểu union như sau:

```
typedef  union <tên kiểu union> {
        <kiểu dl> <tên trường>;
        <kiểu dl> <tên trường>;
        <kiểu dl> <tên trường>;
        …
```

}[<Name>];

Ví dụ: ta có thể định nghĩa kiểu số như sau:

```
typedef  union tagNumber {
        int i;
        long l;
} Number;
```

Việc truy xuất đến một trường trong union được thực hiện hoàn toàn giống như trong struct. Giả sử có biến n kiểu Number. Khi đó, n.i là một số kiểu int còn n.l là một số kiểu long, nhưng cả hai đều dùng chung một vùng nhớ, Vì vậy, khi ta gán: n.i = 0xfd03;
thì giá trị của n.i cũng bị thay đổi (n.i sẽ bằng 3)

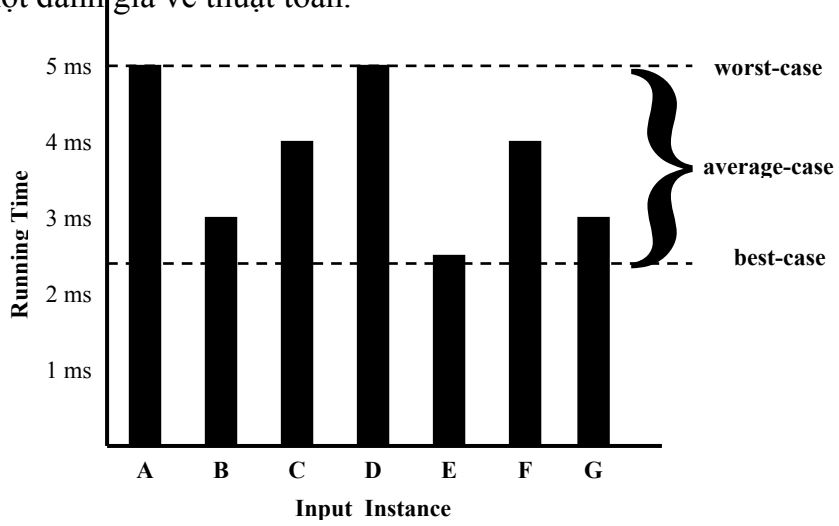
Việc dùng kiểu union rất có lợi khi cần khai báo các cấu trúc dữ liệu mà nội dung của nó thay đổi tùy trạng thái. Ví dụ để mô tả các thông tin về một nhân viên ta có thể khai báo một kiểu dữ liệu như sau:

```
 struct  tagNhanVien
 {      char    HoTen[30];
        int     NamSinh;
          char  NoiSinh[50];
        char    GioiTinh; //0: Nữ, 1: Nam
        char    DiaChi[50];
        char    Ttgd; //0: Không có gia đình, 1: Có gia đình
        union   { char tenVo[30];
                  char tenChong[30];
                }
 } NhanVien;
```

4- Đánh giá độ phức tạp của giải thuật:

Hầu hết các bài toán đều có nhiều thuật toán khác nhau để giải quyết chúng. Như vậy, làm thế nào để chọn được sự cài đặt tốt nhất? Đây là một lĩnh vực được quan tâm nghiên cứu nhiều trong khoa học máy tính. Chúng ta sẽ khảo sát các kết quả nghiên cứu mô tả các tính năng của các thuật toán cơ bản cũng như so sánh các thuật toán đồng thời cũng sẽ khảo sát hướng dẫn tổng quát về phân tích thuật toán.

Khi nói đến hiệu quả của một thuật toán, người ta thường quan tâm đến chi phí cần dùng để thực hiện nó. Chi phí này thể hiện qua việc sử dụng tài nguyên như bộ nhớ, thời gian sử dụng CPU, … Ta có thể đánh giá thuật toán bằng phương pháp thực nghiệm thông qua việc cài đặt thuật toán rồi chọn các bộ dữ liệu thử nghiệm. Thống kê các thông số nhận được khi chạy các dữ liệu này ta sẽ có một đánh giá về thuật toán.



Tuy nhiên, phương pháp thực nghiệm có một số nhược điểm sau khiến nó khó có khả năng áp dụng trên thực tế:

- Do phải cài đặt bằng một ngôn ngữ lập trình cụ thể nên thuật toán sẽ chịu sự hạn chế của ngôn ngữ lập trình này.
- Hiệu quả của thuật toán sẽ bị ảnh hưởng bởi trình độ của người cài đặt.
- Việc chọn được các bộ dữ liệu thử nghiệm đặc trưng cho tất cả tập các dữ liệu vào của thuật toán là rất khó khăn và tốn nhiều chi phí.
- Các số liệu thu nhận được phụ thuộc nhiều vào phần cứng mà thuật toán được thử nghiệm trên đó. Điều này khiến cho việc so sánh các thuật toán khó khăn nếu chúng được thử nghiệm ở những nơi khác nhau.

Vì những lý do trên, người ta đã tìm kiếm những phương pháp đánh giá thuật toán hình thức hơn, ít phụ thuộc môi trường cũng như phần cứng hơn. Một phương pháp như vậy là phương pháp đánh giá thuật toán theo hướng xấp xỉ tiệm cận qua các khái niệm toán học O lớn -O(), o nhỏ - o(), Ω(), ≡()

Thông thường các vấn đề mà chúng ta giải quyết có một "kích thước" tự nhiên (thường là số lượng dữ liệu được xử lý) mà chúng ta sẽ gọi là N. Chúng ta muốn mô tả tài nguyên cần được dùng (thông thường nhất là thời gian cần thiết để giải quyết vấn đề) như một hàm số theo N. Chúng ta quan tâm đến **trường hợp trung bình**, tức là thời gian cần thiết để xử lý dữ liệu nhập thông thường, và cũng quan tâm đến **trường hợp xấu nhất**, tương ứng với thời gian cần thiết khi dữ liệu rơi vào trường hợp xấu nhất có thể có.

Việc xác định chi phí trong trường hợp trung bình thường được quan tâm nhiều nhất vì nó đại diện cho đa số trường hợp sử dụng thuật toán. Tuy nhiên, việc xác định chi phí trung bình này lại gặp nhiều khó khăn. Vì vậy, trong nhiều trường hợp, người ta xác định chi phí trong trường hợp xấu nhất (chặn trên) thay cho việc xác định chi phí trong trường hợp trung bình. Hơn nữa, trong một số bài toán, việc xác định chi phí trong trường hợp xấu nhất là rất quan trọng. Ví dụ, các bài toán trong hàng không, phẫu thuật, …

4.1- Các bước phân tích bài toán:

Bước đầu tiên trong việc phân tích một thuật toán là xác định đặc trưng dữ liệu sẽ được dùng làm dữ liệu nhập của thuật toán và quyết định phân tích nào là thích hợp. Về mặt lý tưởng, chúng ta muốn rằng với một phân bố tùy ý được cho của dữ liệu nhập, sẽ có sự phân bố tương ứng về thời gian hoạt động của thuật toán. Chúng ta không thể đạt tới điều lý tưởng này cho bất kỳ một thuật toán không tầm thường nào, vì vậy chúng ta chỉ quan tâm đến bao đóng của thống kê về tính năng của thuật toán bằng cách cố gắng chứng minh thời gian chạy luôn luôn nhỏ hơn một "chận trên" bất chấp dữ liệu nhập như thế nào và cố gắng tính được thời gian chạy trung bình cho dữ liệu nhập "ngẫu nhiên".

Bước thứ hai trong phân tích một thuật toán là nhận ra các thao tác trừu tượng của thuật toán để tách biệt sự phân tích với sự cài đặt. Ví dụ, chúng ta tách biệt sự nghiên cứu có bao nhiêu phép so sánh trong một thuật toán sắp xếp khỏi sự xác định cần bao nhiêu micro giây trên một máy tính cụ thể; yếu tố thứ nhất được xác định bởi tính chất của thuật toán, yếu tố thứ hai lại được xác định bởi tính chất của máy tính. Sự tách biệt này cho phép chúng ta so sánh các thuật toán một cách độc lập với sự cài đặt cụ thể hay độc lập với một máy tính cụ thể.

Bước thứ ba trong quá trình phân tích thuật toán là sự phân tích về mặt toán học, với mục đích tìm ra các giá trị trung bình và trường hợp xấu nhất cho mỗi đại lượng cơ bản. Chúng ta sẽ không gặp khó khăn khi tìm một chận trên cho thời gian chạy chương trình, vấn đề là phải tìm ra một chận trên tốt nhất, tức là thời gian chạy chương trình khi găp dữ liệu nhập của trường hợp xấu nhất. Trường hợp trung bình thông thường đòi hỏi một phân tích toán học tinh vi hơn trường hợp xấu nhất. Mỗi khi đã hoàn thành một quá trình phân tích thuật toán dựa vào các đại lượng cơ bản, nếu thời gian kết hợp với mỗi đại lượng được xác định rõ thì ta sẽ có các biểu thức để tính thời gian chạy.

Nói chung, tính năng của một thuật toán thường có thể được phân tích ở một mức độ vô cùng chính xác, chỉ bị giới hạn bởi tính năng không chắc chắn của máy tính hay bởi sự khó khăn trong việc xác định các tính chất toán học của một vài đại lượng toán học trừu tượng. Tuy nhiên, thay vì phân tích một cách chi tiết chúng ta thường thích ước lượng để tránh sa vào chi tiết.

4.2- Sự phân lớp các thuật toán:

Như đã được chú ý ở trên, hầu hết các thuật toán đều có một tham số chính là N, thông thường đó là số lượng các phần tử dữ liệu được xử lý mà ảnh hưởng rất nhiều tới thời gian chạy. Tham số N có thể là bậc của một đa thức, kích thước của một tập tin được sắp xếp hay tìm kiếm, số nút trong một đồ thị, v.v… Hầu hết tất cả các thuật toán trong giáo trình này có thời gian chạy tiệm cận tới một trong các hàm sau:

a. **Hằng số**: Hầu hết các chỉ thị của các chương trình đều được thực hiện một lần hay nhiều nhất chỉ một vài lần. Nếu tất cả các chỉ thị của cùng một chương trình có tính chất này thì chúng ta sẽ nói rằng thời gian chạy của nó là hằng số. Điều này hiển nhiên là hoàn cảnh phấn đấu để đạt được trong việc thiết kế thuật toán.

b. **logN**: Khi thời gian chạy của chương trình là logarit tức là thời gian chạy chương trình tiến chậm khi N lớn dần. Thời gian chạy thuộc loại này xuất hiện trong các chương trình mà giải một bài toán lớn bằng cách chuyển nó thành một bài toán nhỏ hơn, bằng cách cắt bỏ kích thước bớt một hằng số nào đó. Với mục đích của chúng ta, thời gian chạy có được xem như nhỏ hơn một hằng số "lớn". Cơ số của logarit làm thay đổi hằng số đó nhưng không nhiều: Khi N là 1000 thì logN là 3 nếu cơ số là 10, là 10 nếu cơ số là 2; khi N là một triệu, logN được nhân gấp đôi. bất cứ khi nào N được nhân đôi, logN tăng lên thêm một hằng số.

c. **N**: Khi thời gian chạy của một chương trình là tuyến tính, nói chung đây là trường hợp mà một số lượng nhỏ các xử lý được làm cho mỗi phần tử dữ liệu nhập. Khi N là một triệu thì thời gian chạy cũng cỡ như vậy. Khi N được nhân gấp đôi thì thời gian chạy cũng được nhân gấp đôi. Đây là tình huống tối ưu cho một thuật toán mà phải xử lý N dữ liệu nhập (hay sản sinh ra N dữ liệu xuất).

d. **NlogN**: Đây là thời gian chạy tăng dần lên cho các thuật toán mà giải một bài toán bằng cách tách nó thành các bài toán con nhỏ hơn, kế đến giải quyết chúng một cách độc lập và sau đó tổ hợp các lời giải. Bởi vì thiếu một tính từ tốt hơn (có lẽ là "tuyến tính logarit"?), chúng ta nói rằng thời gian chạy của thuật toán như thế là "NlogN". Khi N là một triệu, NlogN có lẽ khoảng 20 triệu. khi N được nhân gấp đôi, thời gian chạy bị nhân lên nhiều hơn gấp đôi (nhưng không nhiều lắm).

e. $\mathbf{N^2}$: Khi thời gian chạy của một thuật toán là bậc hai, trường hợp này chỉ có ý nghĩa thực tế cho các bài toán tương đối nhỏ. Thời gian bình phương thường tăng dần lên trong các thuật toán mà xử lý tất cả các phần tử dữ liệu (có thể là hai vòng lặp lồng nhau). Khi n là

một ngàn thì thời gian chạy là 1 triệu. khi N được nhân đôi thì thời gian chạy tăng lên gấp 4 lần.

f. $\mathbf{N^3}$: Tương tự, một thuật toán mà xử lý các bộ ba của các phần tử dữ liệu (có thể là 3 vòng lặp lồng nhau) có thời gian chạy bậc ba và cũng chí ý nghĩa thực tế trong các bài toán nhỏ. Khi N là một trăm thì thời gian chạy là một triệu. Khi N được nhân đôi thì thời gian chạy tăng lên gấp 8 lần.

g. **2N**: Một số ít thuật toán có thời gian chạy lũy thừa lại thích hợp trong một số trường hợp thực tế, mặc dù các thuật toán như thế là "sự ép buộc thô bạo" để giải các bài toán. Khi N là hai mươi thì thời gian chạy là 1 triệu. khi N tăng gấp đôi thì thời gian chạy được nâng lên luỹ thừa hai!

Thời gian chạy của một chương trình cụ thể đôi khi là một hệ số hằng nhân với các số hạng nói trên ("số hạng dẫn đầu") cộng thêm một số hạng nhỏ hơn. Giá trị của hệ số hằng và các số hạng phụ thuộc vào kết quả của sự phân tích và các chi tiết cài đặt. Hệ số của số hạng dẫn đầu liên quan tới số chỉ thị bên trong vòng lặp: Ở một tầng tùy ý của thiết kế thuật toán thì phải cẩn thận giới hạn số chỉ thị như thế. Với N lớn thì các số hạng dẫn đầu đóng vai trò chủ chốt; với N nhỏ thì các số hạng cùng đóng góp vào sự so sánh các thuật toán sẽ khó khăn hơn. Trong hầu hết các trường hợp, chúng ta sẽ gặp các chương trình có thời gian chạy là "tuyến tính", "NlogN", "bậc ba",… với hiểu ngầm là các phân tích hay nghiên cứu thực tế phải được làm trong trường hợp mà tính hiệu quả là rất quan trọng.

4.3- Phân tích trường hợp trung bình

Một tiếp cận trong việc nghiên cứu tính năng của thuật toán là khảo sát **trường hợp trung bình**. Trong tình huống đơn giản nhất, chúng ta có thể đặc trưng chính xác các dữ liệu nhập của thuật toán: ví dụ một thuật toán sắp xếp có thể thao tác trên một mảng N số nguyên ngẫu nhiên, hay một thuật toán hình học có thể xử lý N điểm ngẫu nhiên trên mặt phẳng với các toạ độ nằm giữa 0 và 1. kế đến là tính toán thời gian thực hiện trung bình của mội chỉ thị, và tính thời gian chạy trung bình của chương trình bằng cách nhân tần số sử dụng của mỗi chỉ thị với thời gian cần cho chỉ thị đó, sau cùng cộng tất cả chúng lại với nhau. Có ít nhất ba khó khăn trong cách tiếp cận này như thảo luận dưới đây:

- Trước tiên, là trên một số máy tính rất khó xác định chính xác số lượng thời gian đòi hỏi cho mỗi chỉ thị. Trường hợp xấu nhất thì đại lượng này bị thay đổi và một số lượng lớn các phân tích chi tiết cho một máy tính có thể không thích hợp đối với một máy tính khác. Đây chính là vấn đề mà các nghiên cứu về độ phức tạp tính toán cũng cần phải né tránh.
- Thứ hai, chính việc phân tích trường hợp trung bình lại thường là đòi hỏi toán học quá khó. Do tính chất tự nhiên của toán học thì việc chứng minh các chận trên thì thường ít phức tạp hơn bởi vì không cần sự chính xác. hiện nay chúng ta chưa biết được tính năng trong trường hợp trung bình của rất nhiều thuật toán.
- Thứ ba (và chính là điều quan trọng nhất) trong việc phân tích trường hợp trung bình là mô hình dữ liệu nhập có thể không đặc trưng đầy đủ dữ liệu nhập mà chúng ta gặp trong thực tế. Ví dụ như làm thế nào để đặc trưng được dữ liệu nhập cho chương trình xử lý văn bản tiếng Anh? Một tác giả đề nghị nên dùng các mô hình dữ liệu nhập chẳng hạn như "tập tin thứ tự ngẫu nhiên" cho thuật toán sắp xếp, hay "tập hợp điểm ngẫu nhiên" cho thuật toán hình học, đối với những mô hình như thế thì có thể đạt được các kết quả toán học mà tiên đoán được tính năng của các chương trình chạy trên các ứng dụng thông thường.

TÓM TẮT:

Trong chương này, chúng ta đã xem xét các khái niệm về cấu trúc dữ liệu, kiểu dữ liệu. Thông thường, các ngôn ngữ lập trình luôn định nghĩa sẵn một số kiểu dữ liệu cơ bản. Các kiểu dữ liệu này thường có cấu trúc đơn giản. Để thể hiện được các đối tượng đa dạng trong thế giới thực, chỉ dùng các kiểu dữ liệu này là không đủ. Ta cần xây dựng các kiểu dữ liệu mới phù hợp với đối tượng mà nó biểu diễn. Thành phần dữ liệu luôn là một vế quan trọng trong mọi chương trình. Vì vậy, việc thiết kế cấu trúc dữ liệu tốt là một vấn đề đáng quan tâm.

Vế thứ hai trong chương trình là các thuật toán (thuật giải). Một chương trình tốt phải có các cấu trúc dữ liệu phù hợp với các thuật toán hiệu quả. Khi khảo sát các thuật toán, chúng ta quan tâm đến chi phí thực hiện thuật toán. Chí phí này bao gồm chi phí về tài nguyên và thời gian cần để thực hiện thuật toán. Nếu như những đòi hỏi về tài nguyên có thể dễ dàng xác định thì việc xác định thời gian thực hiện nó không đơn giản . Có một số cách khác nhau để ước lượng khoảng thời gian này. Tuy nhiên, cách tiếp cận hợp lý nhất là hướng xấp xỉ tiệm cận. hướng tiếp cận này không phụ thuộc ngôn ngữ, môi trường cài đặt cũng như trình độ của lập trình viên. Nó cho phép so sánh các thuật toán được khảo sát ở những nơi có vị trí địa lý rất xa nhau. Tuy nhiên, khi đánh giá ta cần chú ý thêm đến hệ số vô hướng trong kết quả đánh giá. Có khi hệ số này ảnh hưởng đáng kể đến chi phí thực của của thuật toán.

Do việc đánh giá chi phí thực hiện trung bình của thuật toán thường phức tạp nên người ta thường đánh giá chi phí thực hiện thuật toán trong trường hợp xấu nhất. Hơn nữa, trong một số thuật toán, việc xác định trường hợp xấu nhất là rất quan trọng.

BÀI TẬP CHƯƠNG 1

Bài tập lý thuyết:

1- Tìm thêm một số ví dụ minh hoạ mối quan hệ giữa cấu trúc dữ liệu và giải thuật.

2- Cho biết một số kiểu dữ liệu được định nghĩa sẵn trong một ngôn ngữ lập trình mà bạn thường sử dụng. Cho biết một số kiểu dữ liệu xây dựng trước này có đủ để đáp ứng mọi yêu cầu về tổ chức dữ liệu không?

3- Một ngôn ngữ lập trình có nên cho phép người sử dụng tự định nghĩa thêm các kiểu dữ liệu có cấu trúc? giải thích và cho ví dụ.

4- Cấu trúc dữ liệu và cấu trúc lưu trữ khác nhau những điểm nào? Một cấu trúc dữ liệu có thể có nhiều cấu trúc lưu trữ được không? Ngược lại một cấu trúc lưu trữ có thể tương ứng với nhiều cấu trúc dữ liệu được không? Cho ví dụ minh hoạ.

5- Giả sử có một bảng giờ tàu cho biết thông tin về các chuyến tàu khác nhau của mạng đường sắt. Hãy biểu diễn các dữ liệu này bằng một cấu trúc dữ liệu thích hợp (file, array, struct, …) sao cho dễ dàng truy xuất giờ khởi hành, giờ đến của một chuyến tàu bất kỳ tại một nhà ga bất kỳ.

Bài tập thực hành

6- Giả sử quy tắc tổ chức quản lý nhân viên của một công ty như sau:

- Thông tin về một nhân viên bao gồm lý lịch và bảng chấm công:

+ Lý lịch nhân viên:

- Mã nhân viên: chuỗi 8 ký tự
- Họ, Tên nhân viên: chuỗi 30 ký tự
- Tình trạng gia đình: 1 ký tự (M=Married, S=Single)
- Số con: số nguyên ≤ 20
- Trình độ văn hoá: chuỗi 2 ký tự

(C1 = cấp 1; C2 = cấp 2; c3 = cấp 3; ĐH = đại học, CH = cao học)

- Lương căn bản: số ≤ 1.000.000

+ Chấm công nhân viên:

- Số ngày nghĩ có phép trong tháng : số ≤ 28
- Số ngày nghĩ không phép trong tháng : số ≤ 28
- Số ngày làm thêm trong tháng : số ≤ 28
- Kết quả công việc : chuỗi 2 ký tự (TO = Tốt; BT = đạt ; KE = Kém)
- Lương thực lĩnh trong tháng : số ≤ 2.000.000

- Quy tắc lĩnh lương:

Lương thực lĩnh = Lương căn bản + Phụ trội

Trong đó nếu:

- số con > 2 : Phụ trội = +5% Lương căn bản
- trình độ văn hoá = CH: Phụ trội = +10% lương căn bản
- làm thêm: Phụ trội = +4% lương căn bản / ngày
- nghĩ không phép : Phụ trội = -5% lương căn bản / ngày

- Chức năng yêu cầu:
  - Cập nhật lý lịch, bảng chấm công cho nhân viên (thêm, xoá, sửa)
  - Xem bảng lương hàng tháng
  - Tìm thông tin của một nhân viên

  Tổ chức cấu trúc dữ liệu thích hợp để biểu diễn các thông tin trên, và cài đặt chương trình theo các chức năng đã mô tả.

Lưu ý:

+ Nên phân biệt thông tin mang tính chất tĩnh (lý lịch) và động (chấm công hàng tháng)

+ Số lượng nhân viên tối đa là 50 người.

Chương 2: CÁC THUẬT TOÁN TÌM KIẾM VÀ SẮP XẾP NỘI

1. Các thuật toán tìm kiếm

1.1. Tìm tuyến tính

1.1.1. Giải thuật

Tìm kiếm tuyến tính là một kỹ thuật tìm kiếm rất đơn giản và cổ điển. Thuật toán tiến hành so sánh giá trị cần tìm với phần tử thứ nhất, thứ hai,..., của dãy số cung cấp cho đến khi gặp được phần tử có khóa cần tìm.

Gọi x là giá trị cần tìm và a là mảng chứa dãy số dữ liệu. Các bước tiến hành như sau:

Bước 1: i=1;

Bước 2 : So sánh a[i] với x, có 2 khả năng :

a[i] = x: Tìm thấy. Dừng.

a[i] != x: Sang bước 3.

Bước 3 : i = i + 1; //xét phần tử kế

Nếu i > N: hết mảng, không tìm thấy. Dừng

Ngược lại: lặp lại bước 2.

1.1.2. Cài đặt

Hàm LinearSearch được cài đặt bên dưới sẽ nhận vào một mảng các số nguyên a và một giá trị cần tìm x. Sau khi thực hiện xong hàm trả về vị trí đầu tiên tìm thấy giá trị x nếu tìm thấy x trong mảng a và trả về giá trị -1 nếu x không có trong mảng.

```
1:   int LinearSearch(int a[ ],int N,int x)
2:   {
3:     int   i = 0;
4:     while  (i < N) && (a[i] != x)
5:          i++;
6:     if (i = = N)
7:          return -1; //Không tìm thấy
8:     else
9:          return i; //Tìm thấy x tại vị trí i
10:  }
```

Trong phần cài đặt trên, chúng ta thấy rằng công việc tìm kiếm chỉ đơn giản thực hiện bằng một vòng lặp để duyệt qua tất cả các phần tử trong mảng. Vòng lặp chỉ kết thúc khi tìm thấy giá trị x trong mảng hoặc đã duyệt hết các phần tử có trong mảng. Như vậy khi kết thúc vòng lặp chỉ số i có 2 khả năng xảy ra:

- i = N có nghĩa là đã duyệt qua phần tử cuối cùng của mảng nhưng vẫn không tìm thấy phần tử nào có giá trị băng với x. Do đó, hàm trả về giá trị -1.
- i < N có nghĩa là giá trị của phần tử thứ i bằng với giá trị x cần tìm. Do đó, hàm trả về giá trị i.

1.1.3. Đánh giá thuật toán

Có thể ước lượng độ phức tạp của giải thuật tìm kiếm qua số lượng các phép so sánh được tiến hành để tìm ra x. Các trường hợp giải thuật tìm tuyến tính có thể có:

Trường hợp	Số lần so sánh	Giải thích
Tốt nhất	1	Phần tử đầu tiên có giá trị là x.
Xấu nhất	N+1	Phần tử cuối cùng có giá trị là x.
Trung bình	(N+1)/2	Giả sử xác xuất các phần tử trong mảng nhận giá trị x là như nhau.

Vậy giải thuật tìm tuyến tính có độ phức tạp O(n).

1.1.4. Nhận xét

Giải thuật tìm kiếm tuyến tính không phụ thuộc vào thứ tự của các phần tử trong mảng. Do đó, đây là phương pháp tổng quát nhất để tìm kiếm trên một dãy số bất kỳ.

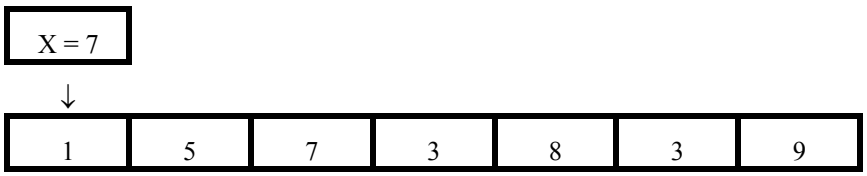
1.1.5. Ví dụ minh họa

Cho dãy số a:

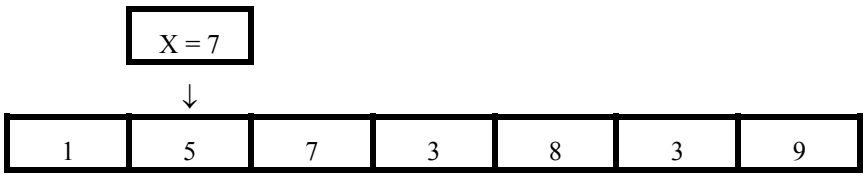
1 5 7 3 8 3 9

Nếu giá trị cần tìm là 7, giải thuật được tiến hành như sau:

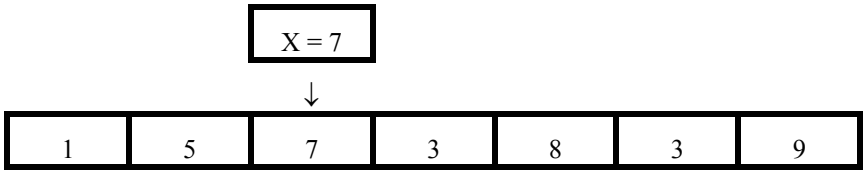
i = 0



i = 1



i = 2



Dừng

1.2. Tìm nhị phân

1.2.1. Giải thuật

Đối với những dãy số đã có thứ tự (giả sử thứ tự tăng), các phần tử trong dãy có quan hệ $a_{i-1} \le a_i \le a_{i+1}$, từ đó kết luận được nếu $x > a_i$ thì x chỉ có thể xuất hiện trong đoạn $[a_{i+1}, a_N]$ của dãy, ngược lại nếu $x < a_i$ thì x chỉ có thể xuất hiện trong đoạn $[a_1, a_{i-1}]$ của dãy. Giải thuật tìm nhị phân áp dụng nhận xét trên đây để tìm các giới hạn phạm vi tìm kiếm sau mỗi lần so sánh x với một phần tử trong dãy.

Ý tưởng của giải thuật là tại mỗi bước tiến hành so sánh x với phần tử nằm ở giữa của dãy tìm kiếm hiện hành, dựa vào kết quả so sánh này để quyết định giới hạn dãy tìm kiếm ở bước kế tiếp là nửa trên hay nửa dưới của dãy tìm kiếm hiện hành.

Gọi x là giá trị cần tìm, a là mảng chứa các giá trị dữ liệu gồm N phần tử đã sắp xếp, left và right là chỉ số đầu và cuối của đoạn cần tìm, middle là chỉ số của phần tử nằm giữa của đoạn cần tìm. Công việc tìm kiếm được tiến hành như sau:

Bước 1: Khởi đầu tìm kiếm trên tất cả các phần tử

left = 1; right = N;

Bước 2: midle = (left+right)/2;

So sánh a[midle] với x, có 3 khả năng :

a[midle] = x : Tìm thấy. Dừng

a[midle] > x : Chuẩn bị tìm tiếp x trong dãy con $a_{left}..a_{midle-1}$: right = midle – 1;

a[midle] < x: Chuẩn bị tìm tiếp x trong dãy con $a_{midle+1}..a_{right}$: left = midle + 1;

Bước 3:

Nếu left <= right : Dãy tìm kiếm hiện hành vẫn còn phần tử. Lặp lại bước 2.

Ngược lại : Dãy tìm kiếm hiện hành hết phần tử. Dừng.

1.2.2. Cài đặt

Tương tự như hàm LinearSearch, hàm BinarySearch cũng nhận vào một mảng nguyên N phần tử và một giá trị cần tìm x. Ngoài ra, để BinarySearch thực hiện đúng mảng nguyên ban đầu phải có thứ tự tăng dần.

```
 1:  int BinarySearch (int a[], int N, int x)
 2:  {
 3:      int left = 0; right = N - 1;
 4:      int middle;
 5:      do
 6:      {
 7:          middle = ( left+right ) / 2;
 8:          if (x = a[middle])
 9:            break; //Da tim thay
10:          else if( x < a[midle] )
11:            right = middle - 1;
12:          else
13:            left = middle + 1;
14:      }
15:      while (left <= right);
```

```
16:         if (left <= right)
17:             return middle; //Tìm thấy x tại vị trí middle
18:         else
19:             return-1;  //Tìm hết dãy mà không có x
20:     }
```

Hàm BinarySearch cũng chỉ đơn giản sử dụng một vòng lặp để duyệt qua các phần tử trong mảng. Tuy nhiên, vòng lặp này sẽ không duyệt qua hết tất cả các phần tử như đối với tìm kiếm tuyến tính. Tư tưởng cài đặt của hàm này cũng giống như đối với LinearSearch, nghĩa là khi kết thúc vòng lặp cũng có 2 khả năng xảy ra:

- Chỉ số left vẫn còn bé hơn hoặc bằng chỉ số right. Điều này có nghĩa là đã có một phần tử nào đó bằng với giá trị x cần tìm, cụ thể là giá trị của phân tử middle. Do đó, hàm sẽ trả về giá trị nằm trong biến middle.
- Chỉ số left đã vượt qua chỉ số right, nghĩa là đã duyệt qua hết các phần tử trong mảng mà không tìm thấy phần tử nào có giá trị bằng x. Do đó, hàm trả về giá trị -1.

1.2.3. Đánh giá thuật toán

Trường hợp giải thuật tìm nhị phân ta có bảng phân tích sau:

Trường hợp	Số lần so sánh	Giải thích
Tốt nhất	1	Phần tử giữa của mảng ban đầu có giá trị x.
Xấu nhất	$\log_2 N$	Phần tử cần tìm nằm ở cuối mảng
Trung bình	$\log_2 N/2$	Giả sử xác xuất các phần tử trong mảng nhận giá trị x là như nhau.

Vậy giải thuật tìm nhị phân có độ phức tạp O(logn).

1.2.4. Nhận xét

Giải thuật tìm nhị phân phụ thuộc vào thứ tự của các phần tử trong mảng để định hướng trong quá trình tìm kiếm, do vậy chỉ áp dụng được cho những dãy đã có thứ tự.

Giải thuật toán tìm nhị phân tiết kiếm thời gian hơn rất nhiều so với giải thuật tìm tuyến tính do $O_{\text{nhị phân}}(\log_2 n) < O_{\text{tuyến tính}}(n)$. Tuy nhiên khi muốn áp dụng giải thuật tìm nhị phân cần phải xét đến thời gian sắp xếp dãy số để thỏa điều kiện dãy số có thứ tự, thời gian này không nhỏ, và khi dãy số biến động cần phải tiến hành sắp xếp lại, … tất cả các nhu cầu đó tạo ra khuyết điểm chính cho giải thuật tìm nhị phân.

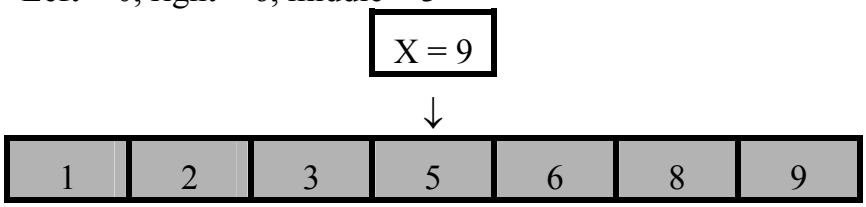
1.2.5. Ví dụ minh họa

Cho dãy số a:

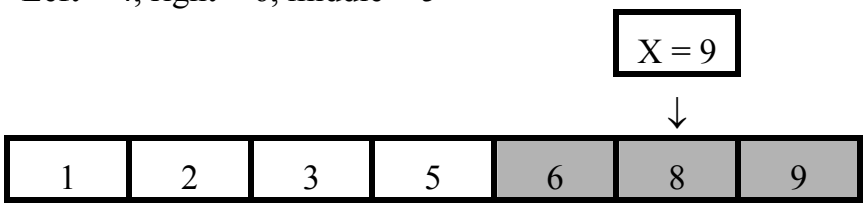
1 2 3 5 6 8 9

Nếu giá trị cần tìm là 9, giải thuật được tiến hành như sau:

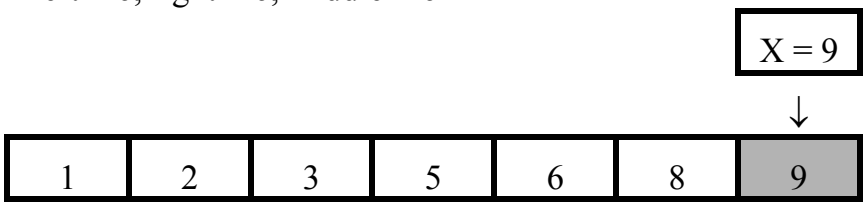
Left = 0, right = 6, middle = 3



Left = 4, right = 6, middle = 5



Left = 6, right = 6, middle = 6



2. Các thuật toán sắp xếp nội

Sắp xếp dãy số $a_1,..., a_n$ là thực hiện việc bố trí lại các phần tử sao cho hình thành được dãy mới $a_{k1}, a_{k2}, \ldots a_{kn}$ có thứ tự (giả sử xét thứ tự tăng) trong đó $a_{ki}<=a_{ki-1}$... Mà để quyết định những tình huống cần thay đổi vị trí các phần tử trong dãy, cần dựa vào kết quả của một loạt phép so sánh. Như vậy, 2 thao tác cơ bản của thuật toán sắp xếp là so sánh và đổi chỗ, khi xây dựng một thuật toán sắp xếp cần chú ý tìm cách giảm thiểu những phép so sánh và đổi chỗ không cần thiết để tăng hiệu quả của thuật toán.

Đối với các dãy số được lưu trữ trong bộ nhớ chính, nhu cầu tiết kiệm bộ nhớ được đặt nặng, do vậy những thuật toán sắp xếp đòi hỏi cấp phát thêm vùng nhớ để lưu trữ dãy kết quả, ngoài vùng nhớ lưu trữ dãy số ban đầu, thường ít được quan tâm. Thay vào đó, các thuật toán sắp xếp trực tiếp trên dãy số ban đầu - gọi là các thuật toán sắp xếp tại chỗ - lại được đầu tư phát triển. Phần này giới thiệu một số giải thuật sắp xếp từ đơn giản đến phức tạp có thể áp dụng thích hợp cho việc sắp xếp nội.

2.1. Các thuật toán cơ bản

2.1.1. Chọn trực tiếp

a- Giải thuật

Ý tưởng của thuật toán chọn trực tiếp mô phỏng một trong những cách sắp xếp tự nhiên nhất thực tế thường được sử dụng: Chọn phần tử nhỏ nhất trong N phần tử ban đầu, đưa phần tử này về vị trí đúng là đầu dãy hiện hành, sau đó không quan tâm đến nó nữa, xem dãy hiện hành chỉ còn N-1 phần tử của dãy ban đầu, bắt đầu từ vị trí thứ 2, lặp lại quá trình trên cho dãy hiện hành chỉ còn 1 phần tử. Dãy ban đầu có N phần tử, vậy tóm tắt ý tưởng thuật toán là thực

hiện N-1 lượt việc đưa phần tử nhỏ nhất trong dãy hiện hành về vị trí đúng ở đầu dãy. Các bước tiến hành như sau :

Bước 1: i =1;

Bước 2: Tìm phần tử a[min] nhỏ nhất trong dãy hiện hành từ a[i] đến a[N].

Bước 3: Hoán vị a[min] và a[i]

Bước 4: Nếu i < N -1 : i = i +1;

lặp lại bước 2

Ngược lại :

N-1 phần tử đã được đưa về đúng vị trí.

Dừng.

b- Cài đặt

Hàm SelectionSort nhận vào một mảng chứa dãy số cần sắp xếp nội dung và tiến hàng sắp xếp ngay trên mảng đã nhập.

```
1:  void SelectionSort(int a[ ],int N)
2:  {
3:      int min; //chỉ số ptử nhỏ nhất trong dãy hiện //hành
4:      int i, j;
5:      for(i = 0; i  < N -1 ; i ++)
6:      {
7:          min = i ;
8:          for(j = i +1; j < N; j++)
9:              //tìm chỉ số ptử nhỏ nhất trong dãy
10:             if (a[j] < a[min])
11:                 min = j;
12:         if (min != i)
13:             HoanVi(a[min],a[i]);
14:     }
15: }
```

c- Nhận xét và đánh giá giải thuật

Đối với giải thuật chọn trực tiếp, có thể thấy rằng ở lượt thứ i, bao giờ cũng cần (n-1) lần so sánh để xác định phần tử nhỏ nhất hiện hành. Số lượng phép so sánh này không phụ thuộc vào tình trạng của dãy số ban đầu, do vậy trong mọi trường hợp có thể kết luận:

Số lần so sánh = n(n-1)/2

Số lần hoán vị lại phụ thuộc vào tình trạng ban đầu của dãy số, ta chỉ có thể ước lược trong từng trường hợp sau:

Trường Hợp	Số lần so sánh	Số lần hoán vị
Tốt nhất	n(n-1)/2	0
Xấu nhất	n(n-1)/2	n(n-1)/2
Trung bình	n(n-1)/2	n(n-1)/4

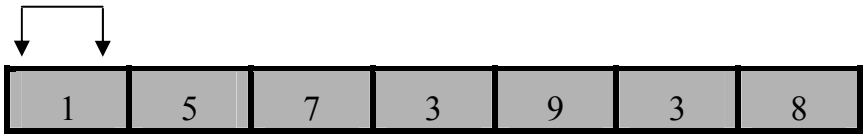
d- Ví dụ minh họa

Cho dãy số a:

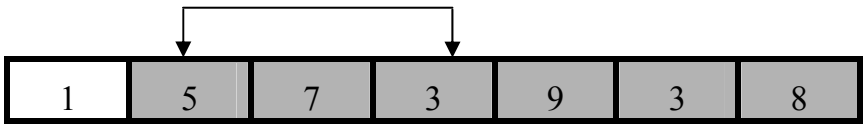
1 5 7 3 9 3 8

Công việc sắp xếp dãy trên bằng thuật toán chọn trực tiếp được tiến hành như sau:

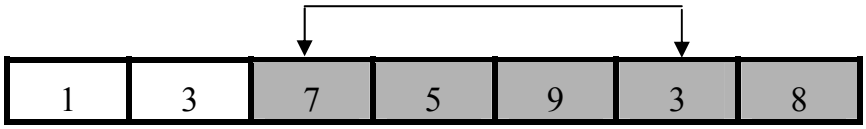
i = 0



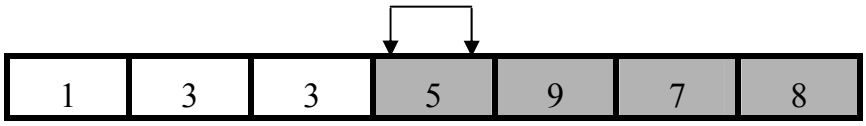
i = 1



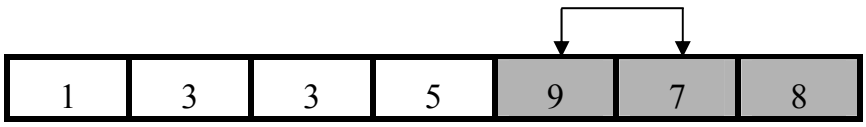
i = 2



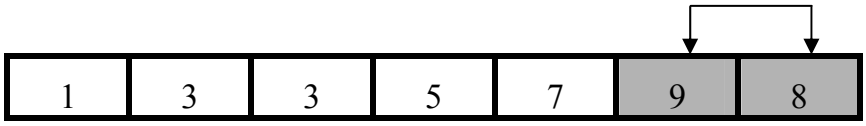
i = 3



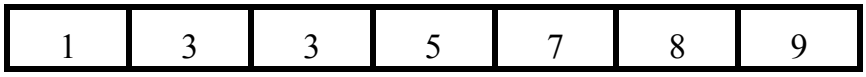
i = 4



i = 5



Dừng



2.1.2. Chèn trực tiếp

a) Giải thuật

Giả sử có một dãy $a_1, a_2,..., a_{i-1}$ đã được sắp xếp, ý tưởng chính của giải thuật sắp xếp bằng cách chèn thêm phần tử a_i vào vị trí thích hợp của đoạn đã được sắp xếp để có dãy mới $a_1, a_2, ..., a_n$. Cho dãy ban đầu $a_1, a_2, ...,a_n$, có thể xem như đã có đoạn gồm một phần tử a_1 đã được sắp, sau đó thêm a_2 vào đoạn a_1 sẽ có đoạn a_1, a_2 đã được sắp; tiếp tục thêm a_3 vào đoạn a_1, a_2 để có đoạn a_1, a_2, a_3 được sắp; tiếp tục cho đến khi thêm xong a_n vào đoạn $a_1, a_2,..., a_{n-1}$ sẽ có dãy $a_1, a_2,..., a_n$ được sắp. Các bước tiến hành như sau:

Bước 1: i = 2;//giả sử có đoạn a[1] đã được sắp.

Bước 2: Tìm vị trí pos thích hợp trong đoạn a[1] đến a[i-1] để chèn a[i] vào.

Bước 3: Dời chỗ các phần tử từ a[pos] đến a[i-1] sang phải một vị trí để dành chỗ cho a[i].

Bước 4: a[pos] = a[i];// có đoạn a[1]..a[i] đã được sắp

Bước 5: i = i + 1;

Nếu i < N : lặp lại bước 2.

Ngược lại :Dừng.

b) - Cài đặt

```
1:  void InsertionSort(int a[ ], intN)
2:  {
3:      int pos, i;
4:      int  x;   //lưu tạm a[i]để hoán vị
5:      for ( int i = 1; i < N; i++ )
6:      {
7:          x = a[ i ];
8:          pos = i - 1;
9:          while (( pos >= 0 ) && (a[pos] > x))
10:         // tìm vị trí chèn x
11:         {
12:                 a[pos + 1] = a[pos];
13:                 pos- -; // trong dãy mới
14:         }
15:         a[pos + 1] = x;   // chèn x vào dãy
16:     }
17: }
```

c) - Nhận xét và đánh giá giải thuật

Khi tìm vị trí thích hợp để chèn a[i] vào đoạn a[1] đến a[i-1], do đoạn đã được sắp nên ta có thể sử dụng giải thuật tìm nhị phân để thực hiện việc tìm vị trí pos. Khi đó, ta có thể cải thiện tốc độ của thuật toán nhờ vào tính hiệu quả của thuật toán tìm kiếm nhị phân.

Đối với giải thuật chèn trực tiếp, các phép so sánh xảy ra trong mỗi vòng lặp **while** tìm vị trí thích hợp **pos**. Và mỗi lần xác định vị trí đang xét không thích hợp, sẽ dời chỗ phần tử

a[pos] tương ứng. Giải thuật thực hiện tất cả N -1 vòng lặp **while**, do số lượng phép so sánh và dời chỗ này phụ thuộc vào tình trạng của dãy số ban đầu, nên chỉ có thể ước lược trong từng trường hợp như sau:

Trường hợp	Số lần so sánh	Số lần hoán vị
Tốt nhất	$\sum_{i=1}^{n-1} 1 = n-1$	$\sum_{i=1}^{n-1} 2 = 2(n-1)$
Xấu nhất	$\sum_{i=1}^{n-1} (i-1) = \frac{(n^2+n)}{2} - 1$	$\sum_{i=1}^{n-1} (i-1) + 2 = \frac{(n^2+n-4)}{2}$
Trung bình	$\frac{(n^2+n-2)}{4}$	$\frac{(n^2+9n-10)}{4}$

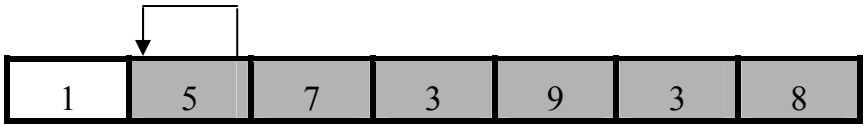
d- Ví dụ minh họa

Cho dãy số a:

1 5 7 3 9 3 8

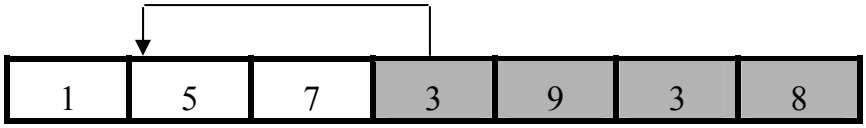
Công việc sắp xếp dãy trên bằng thuật toán chọn trực tiếp được tiến hành như sau:

i = 1

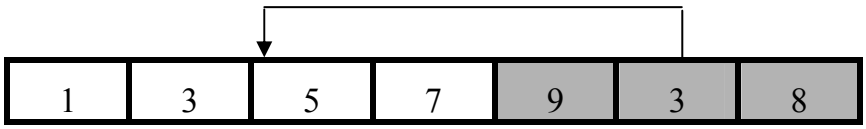


i = 2

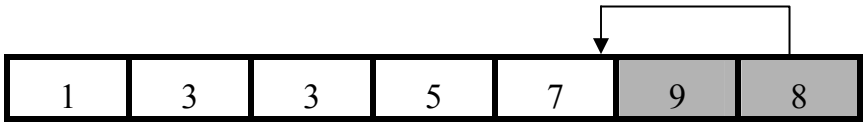
i = 3



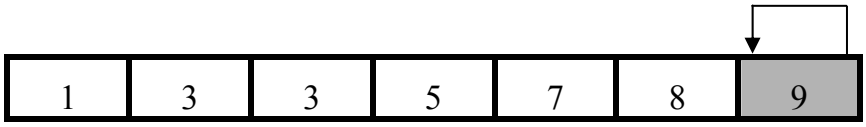
i = 4



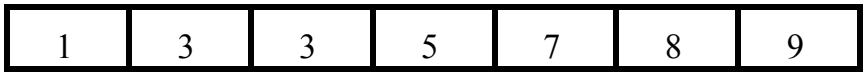
i = 5



i = 6



Dừng



2.1.3. Phương pháp nổi bọt

a) Giải thuật

Ý tưởng của giải thuật là xuất phát từ cuối hoặc đầu dãy và tiến hành đổi chỗ các cặp phần tử kế cận để đưa phần tử nhỏ hoặc lớn hơn về vị trí đúng cao nhất hoặc thấp nhất trong dãy hiện hành. Sau khi đã được chuyển về đúng vị trí thì các phần tử trên sẽ không cần xét đến ở bước tiếp theo. Lặp lại công việc trên cho đế khi không còn phần tử nào để xét.

Bước 1: i = 1

Bước 2: j = N //Duyệt từ cuối dãy đến phần tử thứ i

Trong khi j < i thực hiện:

Nếu a[j] < a[j-1] thì hoán đổi 2 phần tử với nhau

J = j –1

Bước 3: i = i + 1

Nếu i > N – 1: Hết dãy. Dừng

Ngược lại lặp lại bước 2

b) Cài đặt

```
1:  void BubbleSort(int a[], int n)
2:  {
3:    int i, j;
4:    for (i = 0; i < n-1; i++)
5:        for (j = n -1; j > i; j--)
6:          if (a[j] < a[j-1])
7:            HoanVi(a[j],a[j-1]);
      }
```

c) Đánh giá giải thuật

Đối với giải thuật này, số lượng các phép so sánh xảy ra không phụ thuộc tình trạng của dãy số ban đầu, nhưng số lượng phép hoán vị thực hiện tùy thuộc vào kết quả so sánh. Có thể ước lượng trong từng trườn hợp như sau:

Trường hợp	Số lần so sánh	Số lần hoán vị
Tốt nhất	$\sum_{i=1}^{n-1}(n-i+1)=\frac{n(n-1)}{2}$	0
Xấu nhất	$\frac{n(n-1)}{2}$	$\sum_{i=1}^{n-1}(n-i+1)=\frac{n(n-1)}{2}$

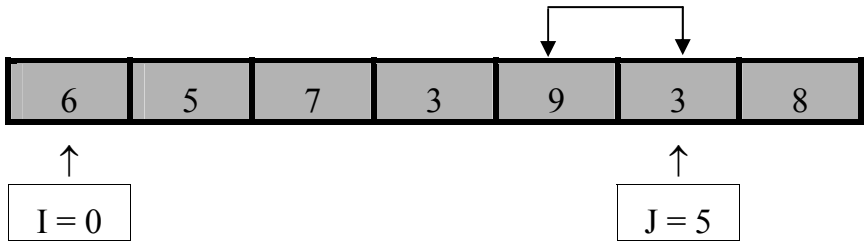
Khi sắp xếp bằng phương pháp BubbleSort thì các phần tử nhỏ được đưa về vị trí đúng rất nhanh, trong khi các phần tử lớn lại được đưa về vị trí đúng rất chậm.

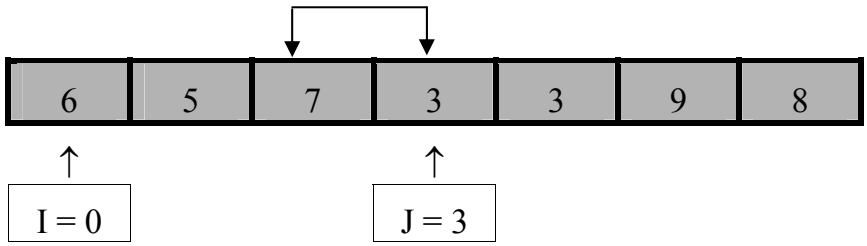
d) Ví dụ minh họa

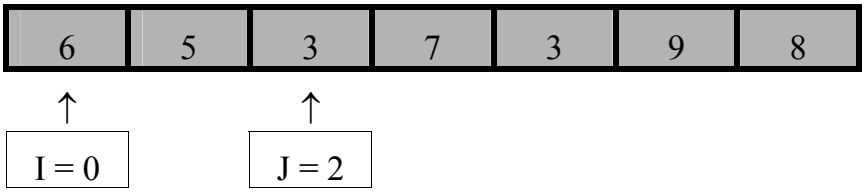
Cho dãy số a:

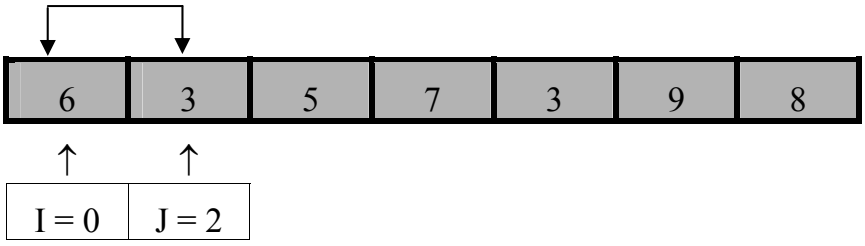
4 5 7 3 9 3 8

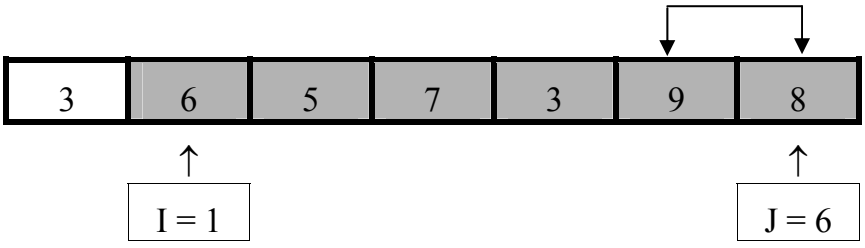
Công việc sắp xếp dãy trên bằng thuật toán nổi bọt được tiến hành như sau:

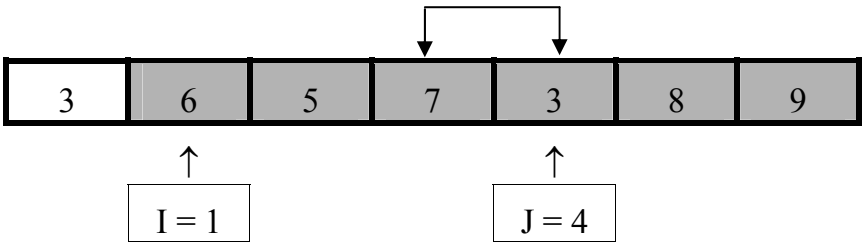


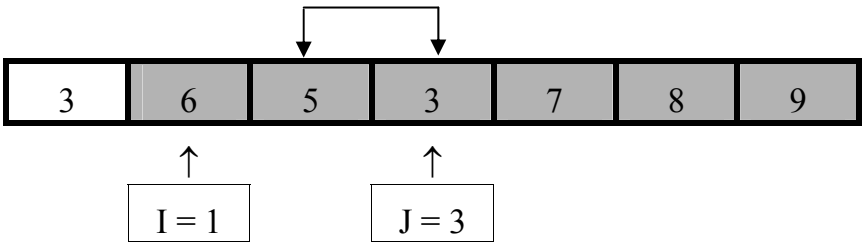


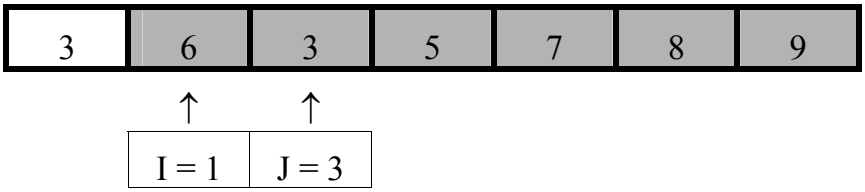

6 5 3 7 3 9 8
I = 0
J = 2

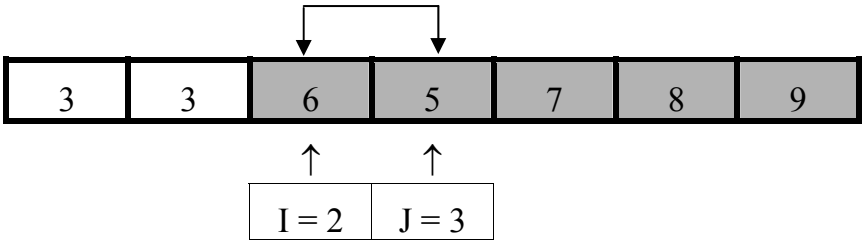

6 3 5 7 3 9 8
I = 0
J = 2


3 6 5 7 3 9 8
I = 1
J = 6

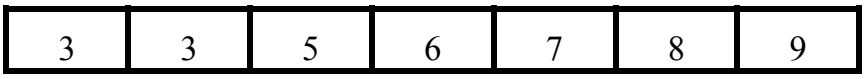

3 6 5 7 3 8 9
I = 1
J = 4


3 6 5 3 7 8 9
I = 1
J = 3





Dừng



2.2. Sắp xếp với độ dài bước giảm dần – Shell Sort

2.2.1. Giải thuật

Giải thuật ShellSort dựa trên ý tưởng sấp xếp các phần tử theo phương pháp chèn nhưng với độ dài bước giảm dần. Ý tưởng của phương pháp sắp xếp là phân chia dãy ban đầu thành những dãy con các phần tử ở cách nhau h vị trí:

Dãy ban đầu : $a_1\ a_2 \ldots a_n$ được xem như sự xen kẽ các dãy con sau :

Dãy con thứ nhất : $a_1\ a_{h+1}\ a_{2h+1} \ldots$

Dãy con thứ hai : $a_2\ a_{h+2}\ a_{2h+2} \ldots$

….

Dãy con thứ h : $a_h\ a_{2h}\ a_{3h} \ldots$

Tiến hành sắp xếp các phần tử trong cùng dãy con sẽ làm cho các phần tử được đưa về vị trí đúng tương đối (chỉ đúng trong dãy con, so với toàn bộ các phần tử trong dãy ban đầu co thể chưa đúng) một cách nhanh chóng, sau đó giảm khoảng cách h để tạo thành các dãy con mới (tạo điều kiện để so sánh một phần tử với nhiều phần tử khác trước đó không ở cùng dãy con với nó) và lại tiếp tục sắp xếp… Thuật toán dừng khi h=1, lúc này bảo đảm tất cả các phần tử trong dãy ban đầu sẽ được so sánh với nhau để xác định trật tự đúng cuối cùng.

Yếu tố quyết định tính hiệu quả của một thuật toán là cách chọn khoảng cách h trong từng bước sắp xếp. Giả sử quyết định sắp xếp k bước, các khoảng cách chọn phải thoả điều kiện:

$h_i > h_{i+1}$ và $h_k = 1$

Tuy nhiên đến nay vẫn chưa có tiêu chuẩn rõ ràng trong việc lựa chọn dãy giá trị khoảng cách tốt nhất, một số dãy được Knuth đề nghị :

$h_i = (h_{i-1} - 1)/3$ và $h_k = 1$, $k = \log_3 n - 1$

hay

$h_i = (h_{i-1} -1)/2$ và $h_k =1$, $k= \log_2-1$

Các bước tiến hành như sau :

Bước 1 **:** Chọn k khoảng cách h[1], h[2],…,h[k] và i=1.

Bước 2 : Phân chia dãy ban đầu thành các dãy con cáhc nhau h[i] khoảng cách. Sắp xếp từng dãy con bằng phương pháp chèn trực tiếp.

Bước 3 : i=i+1;

Nếu I>k : dừng.

Nếu I<=k : lặp lạI bước 2.

2.2.2. Cài đặt

Giả sử chọn được dãy độ dài h[1], h[2],…,h[k], thuật toán ShellSort có thể được cái đặt như sau :

```
void   ShellSort (int a[], int N, int h[], int k)
{
 int step,I,j;
 int x,len;
 for (step=0;step<k, step++)
 {
    len=h[step];
    for (i=len+1;i<N; step++)
    {
        x=a[i];
        j=I-len;
        while (x<a[j])&&(j>-1)
        {
            a[j+len]=a[j];
            j=j-len;
        }
        a[j+len]=x;
    }
 }
}
```

2.2.3. Nhận xét và đánh giá giải thuật

Hiện nay, việc đánh giá giả thuật ShellSort dẫn đến những vấn đề toán học rất phức tạp, thậm chí một số chưa được chứng minh. Tuy nhiên hiệu quả của thuật toán con phụ thuộc vào dãy các độ dài được chọn. Trong trường hợp chọn dãy độ dài theo công thức $h_i = (h_{i-1} -1)/2$ và $h_k =1$, $k= \log_2-1$ thì giải thuật có độ phức tạp $\approx n^{1,2} << n^2$.

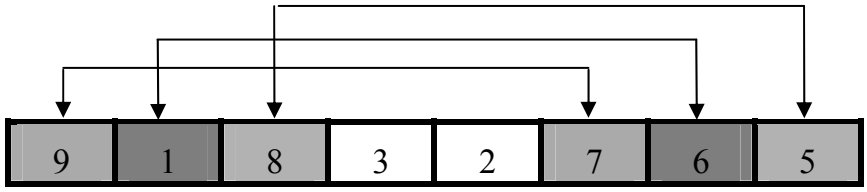
2.2.4.Ví dụ minh họa

Cho dãy số a:

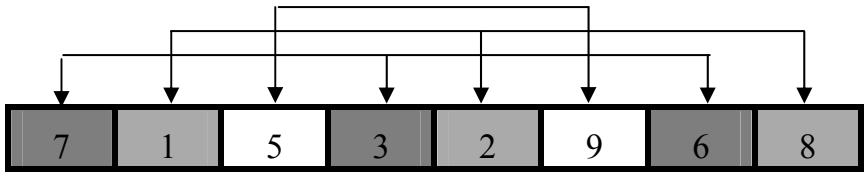
4 5 7 3 9 3 8

Công việc sắp xếp dãy trên bằng thuật toán shell sort với độ dài bước là 5, 3, 1 được tiến hành như sau:

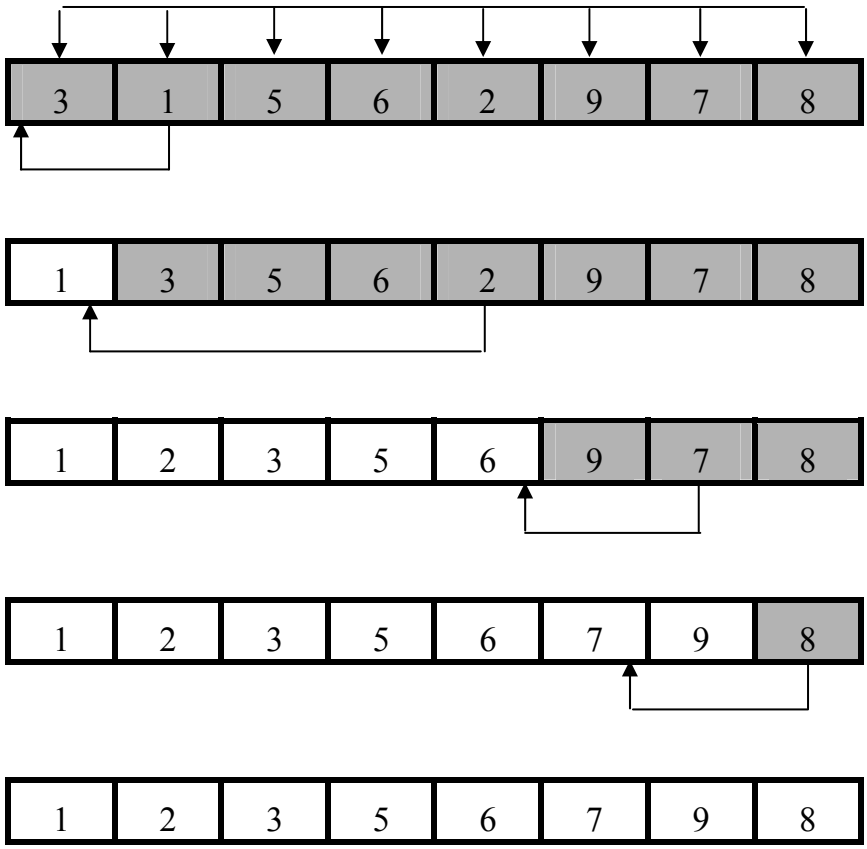
h = 5



h = 3



h = 1



2.3. Sắp xếp dựa trên phân hoạch – Quick Sort

2.3.1. Giải thuật

Để sắp xếp dãy a_1 a_2 …a_n giải thuật Quick Sort dựa trên việc phân hoạch dãy ban đầu thành 3 thành phần :

- Dãy con 1 : Gồm các phần tử a_l …a_i có giá trị không lớn hơn x.

- Dãy con 2 : Gồm các phần tử a_i …a_n có giá trị không nhỏ hơn x.

Với x là giá trị cua rmột phần tử tuỷ ý trong dãy ban đầu. Sau khi thực hiện phân hoạch, dãy ban đầu được chia làm 3 phần :

1. $a_k < x$, vớI k=1..i
2. $a_k = x$, vớI k=i..j
3. $a_k > x$, vớI k=j..N

Trong đó dãy con thứ hai đã có thứ tự, nếu các dãy con 1 và 3 chỉ có 1 phần tử thì chúng cũng đã có thứ tự, khi đó dãy ban đầu đã được sắp. Ngược lại, nếu các dãy con 1 và 3 có nhiều hơn 1 phần tử thì dãy ban đầu chỉ được sắp khi các dãy con 1, 3 có thứ tự. Để sắp xếp dãy con 1 và 3, lần lượt tiến hành phân hoạch từng dãy con theo cùng phương pháp phân hoạch dãy ban đầu vừa trình bày…

Giải thuật để phân hoạch một dãy a_l…a_r thành 2 dãy con:

Bước 1 :

Chọn tuỳ ý một phần tử a[k] trong dãy là giá trị mốc l<=k<=r

X = a[k]; i=l, j=r

Bước 2 :

Phát hiện và hiệu chỉnh cặp phần tử a[i], a[j] nằm sai chỗ:

Bước 2a : Trong khi a[i] <x i++;

Bước 2b : Trong khi a[j]>x j--;

Bước 2c : Nếu i<j Hoán vị (a[i],a[j])

Bước 3 :

- Nếu i<j : Lặp lại bước 2.
- Nếu i>=j : dừng.

Nhận xét

- Về nguyên tắc , có thể chọn giá trị mốc x là một phần tử tuỳ ý trong dãy, nhưng để đơn giản, dễ diễn đạt giải thuật, phần tử có vị trí giữa thường được chọn, khi đó k=(l+r)/2.
- Giá trị mốc x được chọn sẽ tác động đến hiệu quả thực hiện thuật toán vì nó quyết định số lần phân hoạch. Số lần phân hoạch sẽ ít nhất nếu ta chọn được x là phần tử median của dãy. Tuy nhiên, do chi phí xác định phần tử median quá cao nên trên thực tế người ta không chọn phần tử này mà chọn phần tử nằm chính giữa dãy làm mốc với hy vọng nó có thể gần với giá trị median.

Giải thuật để sắp xếp một dãy a_l…a_r

Có thể phát biểu giải thuật sắp xếp QuickSort một cách đệ qui như sau

Bước 1 :

Phân hoạch dẫy a_l…a_r thành các dãy con :

Dãy con 1 : a_l…$a_j < x$

Dãy con 2 : $a_{j+1}…a_{j-1} = x$

Dãy con 3 : $a_j…a_r > x$

Bước 2 : Nếu (l<i) Phân hoạch dãy $a_l…a_j$

Nếu (j<r) Phân hoạch dãy $a_i…a_r$

2.3.2 Cài đặt

```
1:  void QuickSort (int a[], int l, int r)
2:  {
3:    int i,j;
4:    int x;
5:    x=a[(l+r)/2];
6:    i=l;
7:    j=r;
8:    do
9:    {
10:     while (a[i]<x)
11:           i++;
12:     while (a[j]>x)
13:           j--;
14:     if (i<=j)
15:     {
16:           HoanVi(a[i],a[j]);
17:           i++;
18:           j--;
19:     }
20:   } while (i<j);
21:   if (l<i)
22:     QuickSort(a,l,i);
23:   if (j<r)
24:     QuickSort(a,j,r);
25: }
```

2.3.3. Nhận xét và đánh giá giải thuật

Hiệu quả của giải thuật Quick Sort phụ thuộc vào việc chọn giá trị mốc. Trường hợp tốt nhất xảy ra nếu mỗi lần phân hoạch đều chọn được phần tử median làm mốc, khi đó dãy được phân chia thành 2 phần bằng nhau và chỉ cần log(n) lần phân hoạch thì sắp xếp xong. Nhưng nếu mỗi lần phân hoạch lại chọn nhằm phần tử có giá trị cực đại hay cực tiểu làm mốc, dãy sẽ bị phân chia thành 2 phần không đều : một phần chỉ có 1 phần tử, phần còn lại có (n-1) phần tử, do vậy cần phân hoạch n lần mới sắp xếp xong. Ta có bảng tổng kết :

Trường hợp	Độ phức tạp
Trung bình	n*log(n)
Xấu nhất	n^2

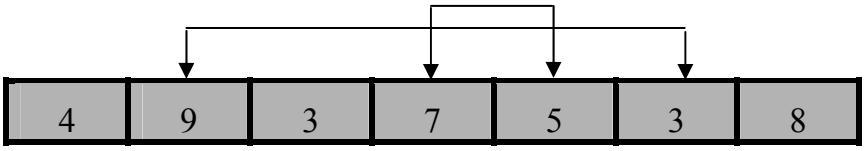
2.3.4.Ví dụ minh họa

Cho dãy số a:

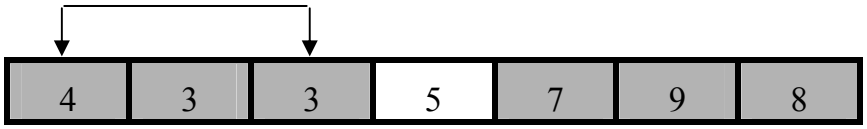
4 9 3 7 5 3 8

Công việc sắp xếp dãy trên bằng thuật toán QuickSort được tiến hành như sau:

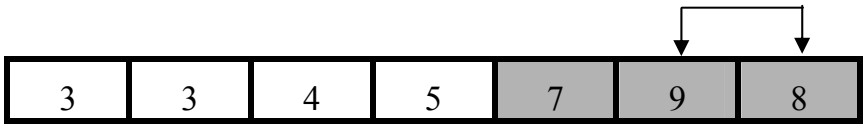
Phân hoạch đoạn l = 0, r = 6, x = a[3] = 7



Phân hoạch đoạn l = 0, r = 2, x = a[1] = 3



Phân hoạch đoạn l = 4, r = 6, x = a[5] = 9



Dừng

2.4. Heap Sort

2.4.1. Định nghĩa cấu trúc dữ liệu Heap

Giả sử xét trường hợp sắp xếp giảm dần, khi đó Heap được định nghĩa là một dãy các phần tử a1, a2, ..., an thỏa các quan hệ :

- ai <= a2i.
- ai <= a2i+1 {(ai, a2i), (ai, a2i+1) là các cặp phần tử liên đới}

Và có các tính chất sau :

- Tính chất 1: Phần tử a1 (đầu Heap) luôn là phần tử nhỏ nhất trong Heap
- Tính chất 2: Cắt bỏ một số phần tử về phía phải của Heap thì dãy con còn lại vẫn là một Heap.

2.4.2. Giải thuật Heapsort

Giải thuật Heapsort trải qua 2 giai đoạn:
Giai đoạn 1:Hiệu chỉnh dãy số ban đầu thành heap;
Giai đoạn 2:Sắp xếp dãy số dựa trên heap:
Bước 1: Đưa phần tử nhỏ nhất về vị trí đúng ở cuối dãy.
Hoán vị(a1,aN);
Bước 2:Loại bỏ phần tử nhỏ nhất ra khỏi dãy:
N = N-1;
Hiệu chỉnh phần còn lại của dãy từ a1, a2,…, an thành một heap.
Bước 3: Nếu N > 1 (heap còn phần tử): lặp lại bước 2.
Ngược lại: Dừng

2.4.3. Cài đặt

Để cài đặt giải thuật Heapsort cần xây dựng các thủ tục phụ trợ:

a) Thủ tục hiệu chỉnh dãy a_1,.., a_r thành heap:

Lần lượt xét các quan hệ của một phần tử liên đới của nó trong dãy là a1 nào đó với các phần tử liên đới của nó trong dãy là a_{2i} và a_{2i+1},nếu vi phạm điều kiện quan hệ của heap,thì đổi chổ a I với phần tử liên đới thích hợp của nó. Lưu ý việc đổi chỗ này có thể gây phản ứng dây chuyền:

```
1:  void    Shift(int a [ ],int l, int r)
2:  {
3:    int  x, l, j, cont;
4:    i = l; j = 2*l;
5:    cont = 1;
6:    //(ai,aj),(ai,aj+1)là các phần tử liên đới
7:    x = a[i ]
8:    while ((j<=r) and (cont))
9:    {
10:   if (j <r) //nếu có đủ 2 phần tử liên đới
11:       //xác định phần tử liên đới nhỏ nhất a[j]
12:
13:       if(a[j]>a[j+1)
14:           j=j+1;
15:   //thỏa quan hệ liên đới ,dừng hiệu chỉnh
16:   if (a[j]>x)
17:       cont = 0;
18:   else
19:   {
20:       a[i] = a[j];
21:       i = j;
22:       //xét tiếp khả năng hiệu chỉnh lan truyền
23:       j=2*i;
24:   }
25:   }
26: }
```

b) Hiệu chỉnh dãy $a_1,..., a_N$ thành heap:

Cho một dãy bất kỳ $a_1,..., a_r$, theo tính chất 3 ta có $a_{n/2+1}, a_{n/2+2},\ldots,a_n$ đã là một heap. Ghép thêm phần tử $a_{n/2}$ vào bên trái heap hiện hành và hiệu chỉnh lại dãy $a_{n/2},\ldots,$ ar thành heap.

```
1:   void  CreateHeap(int a[ ],int N)
2:   {
3:   int   i;
4:   i = N/2;      //a[I] là phần tử ghép thêm
5:   while (i > 1)
6:   {
7:       i--;
8:       Shift(a,i,N);
9:   }
10:  }
```

Khi đó hàm heapsort có dạng như sau:

```
1:  void   Heapsort(int a[ ],int N)
2:  {
3:    int   r;
4:    r = N; //r là vị trí đúng cho phần tử nhỏ nhất
5:    while (r > 1)
6:    {
7:    r--;
8:    Shift(a,1,r);
9:    }
10:  }
```

2.4.4. Đánh giá giải thuật

Việc đánh giá giải thuật Heapsort rất phức tạp, nhưng đã chứng minh được trong trường hợp xấu nhất độ phức tạp gần bằng nlog2n.

TÓM TẮT:

Trong chương này, chúng ta đã xem xét các thuật toán tìm kiếm và sắp xếp thông dụng. Cấu trúc dữ liệu chính để minh hoạ các thao tác này chủ yếu là mảng một chiều. Đây cũng là một trong những cấu trúc dữ liệu thông dụng nhất.

Khi khảo sát các thuật toán tìm kiếm, chúng ta đã làm quen với hai thuật toán. Thuật toán thứ nhất là thuật toán tìm kiếm tuần tự. Thuật toán này có độ phức tạp tuyến tính (O(n)). Ưu điểm của nó là tổng quát và có thể mở rộng để thực hiện các bài toán tìm kiếm đa dạng. Tuy nhiên, chi phí thuật toán khá cao nên ít khi được sử dụng. Thuật toán thứ hai là thuật toán nhị phân tìm kiếm. Thuật toán này có ưu điểm là tìm kiếm rất nhanh (độ phức tạp là $\log_2 N$). Nhưng chỉ có thể áp dụng đối với dữ liệu đã có thứ tự theo khoá tìm kiếm. Do đòi hỏi của thực tế, thao tác tìm kiếm phải nhanh vì đây là thao tác có tần suất sử dụng rất cao nên thuật toán nhị phân tìm kiếm thường được dùng hơn thuật toán tìm tuần tự. Chính vì vậy xuất hiện nhu cầu phát triển các thuật toán sắp xếp hiệu quả.

Phần tiếp theo của chương trình bày các thuật toán sắp xếp thông dụng theo thứ tự từ đơn giản đến phức tạp (từ chi phí cao đến chi phí thấp).

Phần lớn các thuật toán sắp xếp cơ bản dựa trên sự so sánh giá trị giữa các phần tử. Bắt đầu từ nhóm các thuật toán cơ bản, đơn giản nhất. Đó là các thuật toán chọn trực tiếp, chèn trực tiếp, nổi bọt, đổi chỗ trực tiếp. Các thuật toán này đều có một điểm chung là chi phí thực hiện tỷ lệ với n^2.

Tiếp theo, chúng ta khảo sát một số cải tiến của các thuật toán trên. Nếu như các thuật toán chèn nhị phân (cải tiến của chèn trực tiếp), shaker sort (cải tiến của nổi bọt) ; tuy chi phí có ít hơn các thuật toán gốc nhưng chúng vẫn chỉ là các thuật toán thuộc nhóm có độ phức tạp $O(n^2)$, thì các thuật toán shell sort (cải tiến của nhèn trực tiếp), heap sort (cải tiến của chọn trực tiếp) lại có độ phức tạp nhỏ hơn hẳn các thuật toán gốc. Thuật toán shell sort có độ phức tạp $O(n^x)$ với $1<x<2$ và thuật toán heap sort có độ phức tạp $O(n\log_2 n)$.

Các thuật toán Merge sort và Quick sort là những thuật toán thực hiện theo chiến lược chia để trị. Cài đặt chúng tuy phức tạp hơn các thuật toán khác nhưng chi phí thực hiện lại thấp, cả hai thuật toán đều có độ phức tạp $O(n\log_2 n)$. Merge sort có nhược điểm là cần dùng thêm bộ nhớ đệm. Thuật toán này sẽ phát huy tốt ưu điểm của mình hơn khi cài đặt trên các cấu trúc dữ liệu khác phù hợp hơn như danh sách liên kết hay file.

Thuật toán Quick sort, như tên gọi của mình được đánh gía là thuật toán sắp xếp nhanh nhất trong số các thuật toán sắp xếp dựa trên nền tảng so sánh giá trị của các phần tử. Tuy có chi phí trong trường hợp xấu nhất là $O(n^2)$ nhưng trong kiểm nghiệm thực tế, thuật toán Quick sort chạy nhanh hơn hai thuật toán cùng nhóm $O(n\log_2 n)$ là merge sort và heap sort. Từ thuật toán quick sort, ta cũng có thể xây dựng được một thuật toán hiệu quả tìm phần tử trung vị (median) của một dãy số.

Người ta cũng chứng minh được rằng $O(n\log_2 n)$ là ngưỡng chặn dưới của các thuật toán sắp xếp dựa trên nền tảng so sánh giá trị của các phần tử. Để vượt qua ngưỡng này, ta cần phát triển thuật toán mới theo hướng khác các thuật toán trên. Radix sort là một thuật toán như vậy. Nó được phát triển dựa trên sự mô phỏng qui trình phân phối thư của những người đưa thư. Thuật toán này đại diện cho nhóm các thuật toán sắp xếp có độ phức tạp tuyến tính. Tuy nhiên, thường thì các thuật toán này không thích hợp cho việc cài đặt trên cấu trúc dữ liệu mảng một chiều.

Trên thực tế dữ liệu cần thao tác có thể rất lớn do vậy không thông thường thì các dữ liệu được lưu trên bộ nhớ thứ cấp, tức trên các đĩa từ. Việc thực hiện các thao tác sắp xếp trên các

dữ liệu này đòi hỏi phải có các phương pháp khác thích hợp. Tuy nhiên trong khuôn khổ giáo trình này, các thuật toán trên là tương đối khó. Do vật, chúng tôi chỉ giới thiệu qua các phương pháp sắp xếp ngoại như là bài đọc thêm trong phần phụ lục ở cuối sách.

BÀI TẬP CHƯƠNG 2

Bài tập lý thuyết

1- Xét mảng các số nguyên có nội dung như sau:

-9 -9 -5 -2 0 3 7 7 10 15

a- Tính số lần so sánh để tìm ra phần tử X = -9 bằng phương pháp:

- Tìm tuyến tính
- Tìm nhị phân

Nhận xét và so sánh 2 phương pháp tìm nêu trên trong trường hợp này và trong trường hợp tổng quát.

b- Trong trường hợp tìm nhị phân, phần tử nào sẽ được tìm thấy (thứ 1 hay 2)

2- xây dựng thuật toán tìm phần tử nhỏ nhất (lớn nhất) trong một mảng các số nguyên.

3- Một giải thuật sắp xếp được gọi là ổn định (stable) nếu sau khi thực hiện sắp xếp, thứ tự tương đối của các mẫu tin có khoá bằng nhau không đổi. Trong các giải thuật đã trình bày, giải thuật nào là ổn định?

4- Trong 3 phương pháp sắp xếp cơ bản (chọn trực tiếp, chèn trực tiếp, nổi bọt) phương pháp nào thực hiện sắp xếp nhanh nhất với một dãy đã có thứ tự? Giải thích.

5- Cho một ví dụ minh hoạ ưu điểm của thuật toán ShakeSort đối với BubleSort khi sắp xếp một dãy số.

6- Xét bản cài đặt thao tác phân hoạch trong thuật toán QuickSort sau đây:

```
i = 0; j = n-1; x = a[n/2];
do
{  while (a[i] < x) i++;
   while (a[j] > x) j--;
   hoanvi (a[i], a[j]);
}while (i <= j);
```

Có dãy a[0], a[1], ..., a[n-1] nào làm đoạn chương trình trên sai hay không? Cho ví dụ minh hoạ.

7- Hãy xây dựng thuật toán tìm phần tử trung vị (median) của một dãy số a_1, a_2, ..., a_n dựa trên thuật toán QuickSort. Cho biết độ phức tạp của thuật toán này.

Bài tập thực hành

8- Cài đặt các thuật toán tìm kiếm và sắp xếp đã trình bày. Thể hiện trực quan các thao tác của thuật toán. Tính thời gian thực hiện của mỗi thuật toán.

9- Hãy viết hàm tìm tất cả các số nguyên tố nằm trong mảng một chiều a có n phần tử.

10- Hãy viết hàm tìm dãy con tăng dài nhất của mảng một chiều a có n phần tử (dãy con là một dãy liên tiếp các phần tử của a).

11- Cài đặt thuật toán tìm phần tử trung vị (median) của một dãy số bạn đã xây dựng trong bài tập 7.

12- Hãy viết hàm đếm số đường chạy của mảng một chiều a có n phần tử (dãy con là một dãy liên tiếp các phần tử của a).

13- Hãy viết hàm trộn hai mảng một chiều có thứ tự tăng b và c có m và n phần tử thành mảng một chiều a cũng có thứ tự tăng.

14- Hãy cài đặt thuật toán trộn tự nhiên. Thử viết chương trình lập bảng so sánh thời gian thực hiện của thuật toán trộn tự nhiên với trộn trực tiếp và thuật toán Quick sort bằng các thử nghiệm thực tế.

15- Hãy viết chương trình minh hoạ trực quan các thuật toán tìm kiếm và sắp xếp để hỗ trợ cho những người học môn cấu trúc dữ liệu.

Chương 3: DANH SÁCH LIÊN KẾT

1- Giới thiệu

Với các cấu trúc dữ liệu được xây dựng từ các kiểu cơ sở như: kiểu thực, kiểu kí tự,. hoặc từ các cấu trúc đơn giản khác như mẩu tin, tập hợp, mảng,. lập trình viên có thể giải quyết hầu hết các bài toán đặt ra. Các đối tượng dữ liệu được xác định thuộc những kiểu dữ liệu này có đặc điểm chung là không thay đổi được kích thước, cấu trúc trong quá trình sống, do đó thường cứng ngắt, gò bó khiến đôi khi khó diễn tả được thực tế vốn sinh động, phong phú. Các kiểu dữ liệu kể trên được gọi là các kiểu dữ liệu tĩnh.

Nhằm đáp ứng nhu cầu thể hiện sát thực tế, bản chất của dữ liệu cũng như xây dựng các thao tác hiệu quả trên dữ liệu, cần phải tìm cách tổ chức kết hợp dữ liệu với những kích thước linh động hơn, có thể thay đổi kích thước, cấu trúc trong suốt thời gian sống. Các hình thức tổ chức dữ liệu như vậy được gọi là **cấu trúc dữ liệu động.**

2- Kiểu con trỏ

2.1- Biến không động (biến tĩnh, biến nửa tĩnh)

Khi xây dựng chương trình, lập trình viên có thể xác định được ngay những đối tượng dữ liệu luôn cần được sử dụng, thông qua nhu cầu thay đổi về số lượng, kích thước. Do đó có thể xác định cách thức lưu trữ chúng ngay từ đầu. Các đối tượng dữ liệu này sẽ được khai báo như các biến không động. Biến không động là những biến thoả mãn các tính chất sau :

- Được khai báo tường minh.
- Tồn tai khi vào phạm vi khai báo và chỉ mất khi ra khỏi phạm vi này.
- Được cấp phát vùng nhớ trong vùng dữ liệu (Data segment) hoặc Stack (đối với biến nửa tĩnh - các biến cục bộ).
- Kích thước không thay đổi trong suốt chu trình sống.

Do được khai báo tường minh, các biến không động có một định danh đã được kết nối với địa chỉ vùng nhớ lưu trữ biến và được truy xuất trực tiếp thông qua định danh đó.

Ví dụ: int a;

char b[10];

2.2-Kiểu con trỏ

Cho trước kiểu T=<V,O>. Kiểu con trỏ, ký hiệu "T_P", chỉ đến các phần tử có kiểu "T" được định nghĩa :

$T_P = <V_P, O_P>$

Trong đó :

- V_P = {{ các địa chỉ có thể lưu trữ những đối tượng có kiểu T},NULL} (với NULL là một giá trị đặc biệt tượng trưng cho một giá trị không biết hoặc không quan tâm).
- O_P = {các thao tác định địa chỉ của một đối tượng thuộc kiểu T khi biết con trỏ chỉ đến đối tượng đó}.

(Thường gồm các thao tác tạo một con trỏ chỉ đến một đối tượng thuộc kiểu T; huỷ một đối tượng dữ liệu thuộc kiểu T khi biết con trỏ chỉ đến đối tượng đó).

- Nói một cách dễ hiểu, kiểu con trỏ là kiểu cơ sở dùng lưu địa chỉ của một đối tượng khác.
- Biến thuộc kiểu con trỏ T_P là biến mà giá trị của nó là địa chỉ của một vùng nhớ ứng với một biến kiểu T, hoặc là giá trị NULL.

Lưu ý :

Kích thước của biến con trỏ tuỳ thuộc vào quy ước số byte địa chỉ trong từng mô hình bộ nhớ của từng ngôn ngữ lập trình cụ thể.

3- Danh sách

Cho T là một kiểu được định nghĩa trước, kiểu danh sách T_X gồm các phần tử thuộc kiểu T được định nghĩa là:

$$T_X = < V_X , O_X >$$

Trong đó :

V_X = { tập hợp các thứ tự gồm một số biến động các phần tử kiểu T }.

O_X = { tạo danh sách; tìm 1 phần tử trong danh sách; chèn 1 phần tử vào danh sách; huỷ 1 phần tử khỏi danh sách; liệt kê danh sách, sắp xếp danh sách.}.

3.1. Danh sách đơn

3.1.1Tổ chức danh sách đơn

Mỗi phần tử của danh sách đơn là một cấu trúc chứa 2 thành phần chính:

- Thành phần Info: lưu trữ các thông tin về bản thân phần tử.

- Thành phần Next: lưu trữ địa chỉ của phần tử kế tiếp trong danh sách, hoặc lưu trữ giá trị NULL nếu là phần tử cuối danh sách.

Ta có định nghĩa tổng quát :

```
typedef struct tagNode
 {
        Data Info;//Data là kiểu đã được định nghĩa
        struct tagNode* Next;//Con trỏ chỉ đến Node
}Node;
```

Một phần tử trong danh sách đơn là một biến động sẽ được yêu cầu cấp phát khi cần. Và danh sách đơn chính là sự liên kết các biến động này với nhau, nhờ vậy đạt được sự linh động khi thay đổi số lượng các phần tử.

Nếu biết được địa chỉ của phần tử đầu tiên trong danh sách đơn thì có thể dựa vào thông tin Next của nó để truy xuất đến phần tử thứ hai của xâu, và lại dựa vào thông tin Next của phần tử thứ hai để truy xuất phần tử thứ ba. Nghĩa là để quản lý một xâu đơn chỉ cần biết địa chỉ

phần tử đầu xâu. Thường một con trỏ Head sẽ được dùng để lưu trữ địa chỉ phần tử đầu xâu, nên ta gọi Head là đầu xâu.

Ta có thể khai báo :

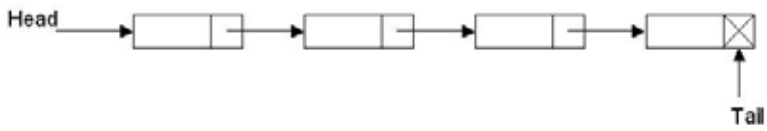
Node *Head.

Tuy về nguyên tắc chỉ cần quản lý xâu thông qua đầu xâu Head, nhưng thực tế có nhiều trường hợp cần làm việc với phần tử cuối cùng, khi đó mỗi lần muốn xác định phần tử cuối cùng lại phải duyệt từ đầu xâu. Để tiện lợi, có thể sử dụng thêm một con trỏ Tail giữ địa chỉ phần tử cuối xâu.

Ta khai báo như sau :

Node *Tail.

Lúc này ta có xâu đơn :



3.1.2. Các thao tác cơ bản trên danh sách đơn

Giả sử có định nghĩa :

```
typedef struct tagNode
{
        Data Info;
        struct tagNode *Next;
}Node;
typedef Node *LIST;
LIST Head, Tail;
Node *new_element;
Data x;
```

Và đã xây dựng được thủ tục Getnode để tạo ra một phần tử cho danh sách :

```
Node *  GetNode(Data x)
{
        Node *p;
        p = (Node*)malloc(sizeof(Node));
        if (p==NULL)
        {
                print(" Không đủ bộ nhớ.");
                exit(1);
        }
```

```
    p->Info=x;
    p->Next=NULL;
    return p;
}
```

Qui ước gọi danh sách đơn với đầu xâu Head là xâu Head.

Phần tử do new_element giữ địa chỉ, tạo bởi câu lệnh :

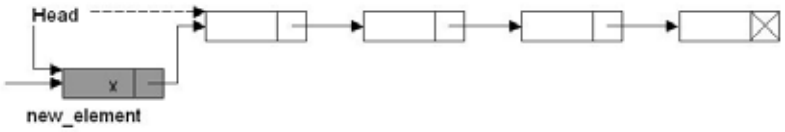
```
new_element = GetNode(x);
```

được gọi là new_element.

a) Chèn một phần tử vào danh sách

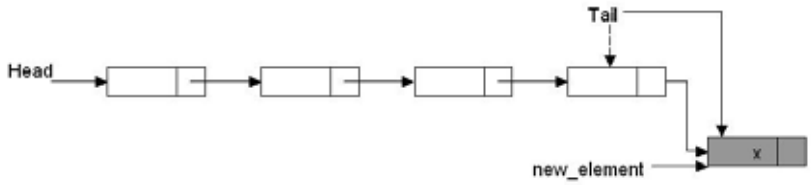
Có 3 cách chèn new_element vào xâu Head

Cách 1 : Chèn vào đầu danh sách



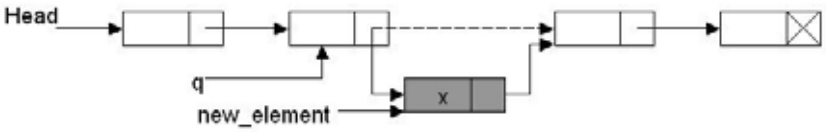

```
 1:  void InsertFirst(LIST &Head, LIST &Tail, Node *new_element)
 2:  {
 3:    if (Head==NULL)
 4:    {
 5:        Head=new_lement;
 6:        Tail=Head;
 7:    }
 8:    else
 9:    {
10:        new_element->Next=Head;
11:        Head=new_element;
12:    }
13:  }
```

Cách 2 : Chèn vào cuối danh sách




```
1:  void InsertEnd(LIST &Head, LIST &Tail, Node *new_element)
2:  {
3:    if (Head==NULL)
4:    {
5:        Head=new_element;
6:        Tail=new_element;
7:    }
8:    else
9:    {
10:       Tail->Next=new_element;
11:       Tail=new_element;
12:   }
13: }
```

Cách 3 : Chèn vào danh sách sau một phần tử q




```
1:  void InsertAfter(LIST &Head, LIST &Tail, Node *q, Node *new_element)
2:  {
3:    if (q!=NULL)
4:    {
5:        new_element->Next=q->Next;
6:        q->Next=new_element;
7:        if (q==Tail)
8:            Tail=new_element;
9:    }
10: }
```

b) Tìm một phần tử trong danh sách đơn

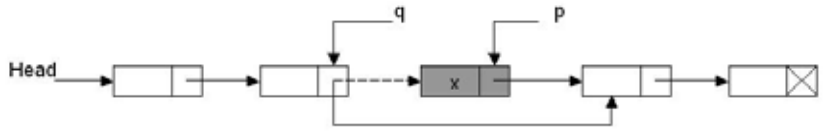
Thuật toán :

Danh sách liên kết đơn đòi hỏi truy xuất tuần tự, do đó áp dụng thuật toán tìm tuyến tính để xác định phần tử trong danh sách có khoá k. Sử dụng một con trỏ phụ trợ p để lần lượt trỏ đến các phần tử trong danh sách. Thuật toán được cài đặt như sau :

```
1:  Node* Search(Node *Head, Data x)
2:  {
3:    Node *p;
4:    p=Head;
5:    while (p->Info!=x)&&(p!=NULL)
6:        p=p->Next;
7:    return p;
8:  }
```

c) Huỷ một phần tử khỏi danh sách

Huỷ một phần tử sau phần tử p




```
1:  void DeleteAfter(LIST &Head, LIST &Tail, Node *q)
2:  {
3:    Node *p;
4:    if (q!=NULL)
5:    {
6:        p=q->Next;
7:        if (p!=NULL)
8:        {
9:            if (p==Tail)
10:               Tail=q;
11:           q->Next=p->Next;
12:           delete p;
13:       }
14:   }
15: }
```

Huỷ một phần tử có khoá k

Thuật toán

Bước 1: p=Search(Head,k); // p là phần tử có khoá k cần huỷ.

Bước 2: Nếu (p!=NULL) thì // giả sử có thủ tục SearchPre để tìm phần tử q đứng trước p trong xâu.

Bước 2.1: q=SearchPre(Head,p);

Bước 2.2: Nếu q=Head thì // q là đầu xâu

Bước 2.2.1: Head=p->Next;

Bước 2.2.2: free(p);

ngược lại

Bước 2.2.3 DeleteAfter(q);

d) Duyệt danh sách

Duyệt danh sách là thao tác thường được thực hiện khi có nhu cầu xử lý các phần tử của danh sách theo cùng một cách thức hoặc khi cần lấy thông tin tổng hợp từ các phần tử của danh sách. Như là :

- Đếm các phần tử của danh sách.
- Tìm tất cả các phần tử thoả điều kiện.
- Huỷ toàn bộ danh sách (và giải phóng bộ nhớ).

Để duyệt danh sách (và xử lý từng phần tử) ta thực hiện như sau :

```
1:    void ProcessList(LIST &Head)
2:    {
3:        Node *p;
4:        p=Head;
5:        while (p!=NULL)
6:        {
7:          ProcessNode(p); //xử lý cụ thể tuỳ trường //hợp
8:          p=p->Next;
9:        }
10:   }
```

e) Sắp xếp danh sách

Một danh sách có thứ tự (danh sách được sắp) là một danh sách mà các phần tử của nó được sắp xếp theo một thứ tự nào đó dựa trên một trường khoá. Để sắp xếp một danh sách, ta có thể thực hiện một trong hai phương án sau :

Phương án 1 : Hoán vị nội dung các phần tử trong danh sách (thao tác trên vùng Info).

Với phương án này, có thể chọn một trong những thuật toán sắp xết đã biết để cài đặt lại trên xâu như thực hiện trên mảng, điểm khác biệt duy nhất là cách thức truy xuất đến các phần tử trên xâu thông qua liên kết thay vì chỉ số như trên mảng. Do dựa trên việc hoán vị nội dung

củấcc phần tử, phương pháp này đòi hỏi sử dụng thêm vùng nhớ trung gian nên chỉ thích hợp với các xâu có các phần tử cácthành phần Info kích thước nhỏ. Hơn nữa, số lần hoán vị có thể lên đến bậc n^2 với xâu có n phần tử, không tận dụng được các ưu điểm của xâu.

Ví dụ: Cài đặt thuật toán sắp xếp **Chọn lựa trực tiếp** trên xâu:

```
1:  void ListStraightSelection(LIST &Head)
2:  {
3:    Node *min;
4:    Node *p,*q;
5:    p=Head;
6:    while (p!=NULL)
7:    {
8:        q=p->Next;
9:        min=p;
10:       while (q!=NULL)
11:       {
12:           if (q->Info<min->Info) min=q;
13:           q=q->Next;
14:       }
15:           Hoanvi(min->Info,p->Info);
16:           p=p->Next;
17:   }
18: }
```

Phương án 2 : Thay đổi các mối kiên kết (thao tác trên vùng Next).

Tạo một danh sách mới là danh sách có thứ tự từ danh sách cũ (đồng thời huỷ danh sách cũ). Giả sử danh sách mới sẽ được quản lý bằng con trỏ đầu xâu Result, ta có thuật toán như sau :

B1: Khởi tạo danh sách mới Result là rỗng;

B2: Tìm trong danh sách cũ Head phần tử min là phần tử nhỏ nhất;

B3: Tách min khỏi danh sách Head;

B4: Chèn min vào cuối danh sách Result;

B5: Lặp lại bước 2 khi chưa hết danh sách Head;

3.2. Danh sách vòng

3.2.1. Tổ chức danh sách vòng

Danh sách vòng là một danh sách đơn mà phần tử cuối danh sách trỏ tới phần tử đầu danh sách. Để biểu diễn, ta có thể sử dụng các kỹ thuật biểu diễn như danh sách đơn.

3.2.2. Các thao tác trên danh sách vòng

a) Tìm phần tử trên danh sách vòng

Danh sách vòng không có phần tử đầu danh sách rõ ràng, nhưng ta có thể đánh dấu một phần tử bất kỳ trên danh sách để kiểm tra việc duyệt đã qua hết các phần tử của danh sách hay chưa.

```
1:  Node*  Search(Node *Head, Data x)
2:  {
3:    Node *p;
4:    Head = p;
5:    do
6:    {
7:        if (p->Info==x) return p;
8:        p=p->Next;
9:    }
10:   while (p!=Head);
11:   return p;
12: }
```

b) Thêm phần tử newelement vào bên phải nút q

Giả sử phần tử newelement cần thêm vào xâu đã được được cấp phát bộ nhớ và gán nội dung.

```
1:  void InsertRight(Node *q, Node *newelement)
2:  {
3:    if (q==NULL)
4:    {
5:        q=newelement;
6:        newelement->Next=q;
7:    }
8:    else
```

```
 9:      {
10:          newelement->Next=q->Next;
11:          q->Next=newelement;
12:      }
13:  }
```

Lưu ý :

Đối với danh sách vòng, có thể xuất phát từ một phần tử bất kỳ để duyệt toàn bộ danh sách.

3.3. Danh sách kép

3.3.1. Tổ chức lưu trữ

Danh sách kép là danh sách mà mỗi phần tử trong danh sách có thể kết nối với 1 phần tử đứng trước và 1 phần tử đứng sau nó.

Các lệnh sau định nghĩa một danh sách kép đơn giản trong đó dùng 2 con trỏ : Pre liên kết với phần tử đứng trước và Next, như thường lệ, liên kết với phần tử đứng sau :

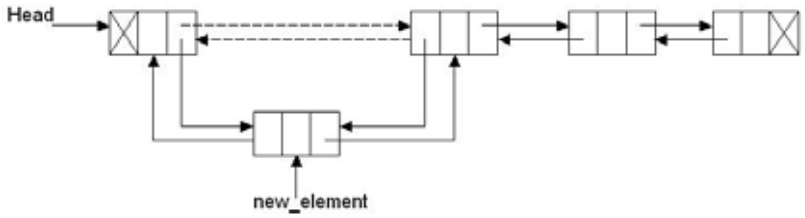
```
typedef struct tagDNode
{
        Data Info;
        struct tagDNode *Pre;
        struct tagDNode *Next;
} DNode;
```

Trong đó, thủ tục khởi tạo một phần tử cho xâu kép được viết như sau :

```
DNode* GetDNode(Data x)
{
    DNode *p;
    p=(DNode*)malloc(sizeof(DNode));
    if (p==NULL)
    {
        puts("Không đủ bộ nhớ.");
        exit(1);
    }
     p->Info=x;
    p->Pre=NULL;
    p->Next=NULL;
    return p;
}
```

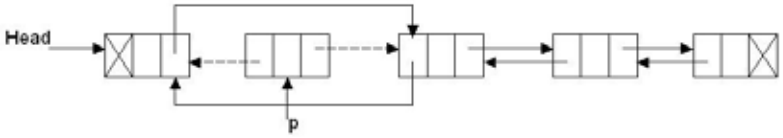
3.3.2. Các thao tác trên danh sách kép

a) Thêm phần tử vào danh sách sau phần tử q




```
1:  void InsertAfter(DNode *q, DNode *newelement)
2:  {
3:    if (q==NULL)
4:      q=newelement;
5:    else
6:    {
7:      newelement->Next=q->Next;
8:      if (q->Next!=NULL)
9:        q->Next->Pre=newelement;
10:     q->Next=newelement;
11:     newelement->Pre=q;
12:   }
13: }
```

b) Huỷ 1 phần tử p của danh sách kép




```
1:  void DeleteAfter(DNode *Head, DNode *p)
2:  {
3:    if (p->Pre==NULL)
4:      Head=p->Next;
5:    else
6:    {
7:      if (p->Next==NULL)
8:        p->Pre->Next=NULL;
9:      else
10:     {
```

```
11:            p->Pre->Next=p->Next;
12:            p->Next->Pre=p->Pre;
13:          }
14:       }
15:       free(p);
16:    }
```

3.4. Danh sách có nhiều mối liên kết

Danh sách có nhiều mối liên kết là danh sách mà mỗi phần tử có nhiều khoá và chúng được liên kết với nhau theo từng loại khoá. Danh sách có nhiều mối liên kết thường được sử dụng trong các ứng dụng quản lý một cơ sở dữ liệu lớn với nhu cầu tìm kiếm dữ liệu theo từng khoá khác nhau.

Để quản lý danh mục điện thoại thuận tiện cho việc in danh mục theo những trình tự khác nhau: tên khách hàng tăng dần, số điện thoại tăng dần, thời gian lắp đặt giảm dần. Ta có thể tổ chức dữ liệu theo dạng sau : một danh sách với 3 mối liên kết: một cho họ tên khách hàng, một cho số điện thoại và một cho thời gian lắp đặt.

Các thao tác trên một danh sách đa liên kết được tiến hành tương tự như trên danh sách đơn nhưng được thực hiện làm nhiều lần và mỗi lần cho một liên kết.

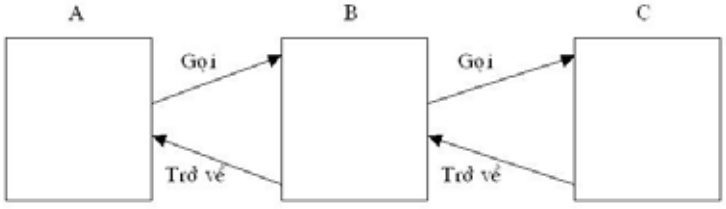
4. Stack

4.1. Định nghĩa

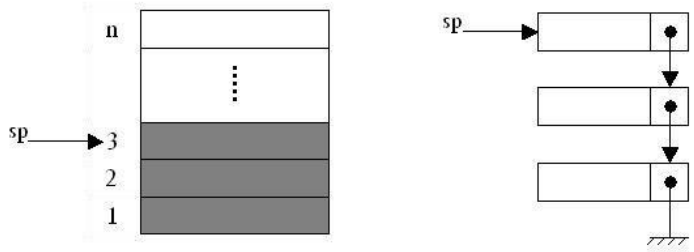
Stack là một danh sách mà 2 phép thêm và loại bỏ đều được thực hiện trên một đầu.

Như vậy, phần tử nào thêm vào sau sẽ bị loại bỏ trước, nên Stack còn được gọi là danh sách LIFO (Last In First Our list) hoặc Pushdown list.

Stack có nhiều ứng dụng. Ví dụ: chương trình A gọi chương trình B, chương trình B gọi chương trình C. Khi chương trình C được thực hiện xong thì sự điều khiển chương trình sẽ trở về thực hiện chương trình B, rồi khi chương trình B được thực hiện xong thì sự điều khiển chương trình sẽ trở về thực hiện chương trình A. Như vậy chương trình B được gọi sau sẽ được trở về thực hiện trước chương trình A. Đó là nhờ điểm nhập (entry point) trở về của các chương trình được chứa trong Stack.



Stack có thể được tổ chức theo dạng mảng hoặc theo danh sách liên kết. Vì phép thêm vào và phép loại bỏ chỉ được thực hiện ở cùng một đầu nên ta chỉ cần một chỉ điểm gọi là con trỏ của satck (stack pointer).



4.2. Biểu diễn Stack dùng mảng

4.2.1. Tổ chức dữ liệu

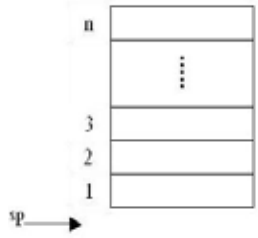
Ta định nghĩa Stack S là mộ mảng các phần tử và một biến sp dùng làm con trỏ Stack.

```
1:  #define MAX 30
2:  typedef struct
3:  {
4:          Data Info;
5:  }item;
6:  item s[MAX];
7:  int sp;
```

4.2.2. Các thao tác trên stack

a) Khởi tạo stack

Khi khởi tạo, stack là rỗng, ta cho sp = -1.

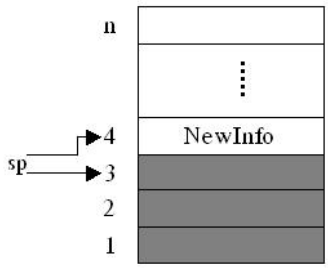


Cài đặt

```
1:  void Initialize()
2:  {
3:    sp = -1;
4:  }
```

b) Thêm một phần tử mới vào stack

Giả ta cần chứa nội dung của NewItem vào stack.

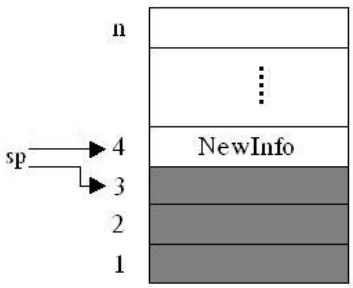


Hàm push trả về giá trị -1 nếu stack bị đầy hoặc trả về vị trí của phần tử mới thêm vào stack.

Cài đặt

```
1:    int push(item NewItem)
2:    {
3:    int kq;
4:      if (sp < (n –1))
5:      {
6:        sp++;
7:        s[sp].Info = item.Info;
8:        kq = sp;
9:      }
10:     else
11:       kq = -1;
12:     return kq;
13:   }
```

c) Lấy một phần tử ra khỏi stack



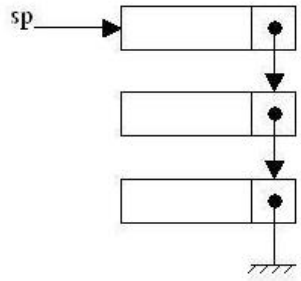
Hàm pop trả về giá trị -1 nếu stack rỗng ngược lại trả về giá trị 1 cùng thông tin đang chứa trong Stack thông qua biến i.

Cài đặt :

```
1:   int pop(item &i)
2:   {
3:     int kq;
4:     if (sp >= 0)
```

```
 5:      {
 6:        i.Info = s[sp].Info;
 7:        sp--;
 8:        kq = 1;
 9:      }
10:      else
11:        kq = -1;
12:      return kq;
13:    }
```

4.3. Stack được tổ chức theo danh sách liên kết



4.3.1. Tổ chức dữ liệu

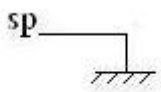
Stack là một danh sách liên kết được khai báo như sau:

```
1: typedef struct tagItem
2: {
3:    Data Info;
4:    struct tagItem *Next;
5: }item;
6: item* sp;
```

4.3.2. Các thao tác trên Stack

a) Khởi tạo stack

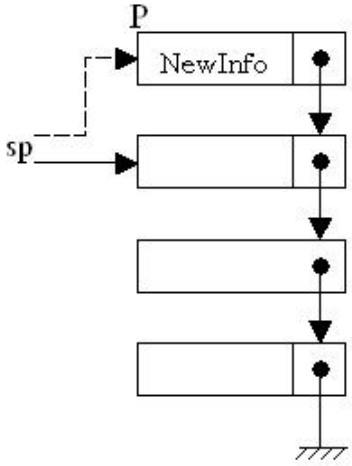
Khi khởi tạo, stack là rỗng, ta cho sp = NULL.



Cài đặt

```
1:  void Initialize()
2:  {
3:    Sp = NULL;
4:  }
```

b) Thêm một phần tử vào stack

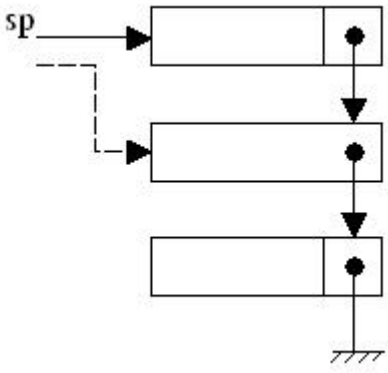


Hàm push tiến hành đưa giá trị Info vào trong Stack. Nếu thực hiện thành công hàm trả về 1, ngược lại hàm trả về 0.

Cài đặt

```
1:  int push(Data NewInfo)
2:  {
3:    item* i;
4:    i = (item*)malloc(sizeof(item));
5:    if (i == NULL)
6:      return 0;
7:    i→Info = NewInfo;
8:    i→Next = sp;
9:    sp = i;
10:   return 1;
11: }
```

c) Lấy một phần tử ra khỏi Stack



Hàm pop tiến hành lấy giá trị hiện tại trong Stack đặt vào biến Info và trả về gía trị 1 nếu trong Stack có phần tử, ngược lại trả về giá trị 0.

Cài đặt

```
1:  int pop(Data& Info)
2:  {
3:    item* i;
4:    if (sp == NULL)
5:      return 0;
6:    Info = sp→Info;
7:    i = sp;
8:    sp = sp→Next;
9:    free(i);
10:   return 1;
11: }
```

5. Hàng đợi (Queue)

5.1. Định nghĩa

Hàng đợi là một danh sách mà phép thêm vào được thực hiện ở đầu này và loại bỏ được thực hiện ở đầu kia.

Như vậy, phần tử nào vào trước sẽ được loạI bỏ trước, phần tử nào vào sau sẽ được loại bỏ sau, nên hang đợi còn được gọi là danh sách FIFO (First In First Out list).

5.2. Cài đặt hàng đợi dùng mảng

5.2.1. Tổ chức dữ liệu

Để có thể cài đặt hàng đợi bằng cấu trúc mảng, ta khai báo như sau:

```
1:  #define MAX 30
2:  typedef struct tagItem
3:  {
4:    Data Info;
5:  }Item;
6:  item queue[MAX];
7:  int head;
```

Biến head chính là vị trí của phần tử sẽ được lấy ra.

5.2.2. Các thao tác trên hàng đợi

a) Khởi tạo hàng đợi

Khi khởi tạo, hàng đợi là rỗng, ta cho head bằng -1.

Cài đặt

```
1:  void Initialize()
```

```
2:  {
3:     head = -1;
4:  }
```

b) Thêm một phần tử vào hàng đợi

Cài đặt

```
1:  int Insert(Data& i)
2:  {
3:     int i;
4:     if (head = n-1) //hàng bị đầy
5:        return –1;
6:     head++;
7:     //Doi cac phan tu len 1 don vi
8:     for (i = head; i > 0; i--)
9:        q[i] = q[i-1];
10:    //Them phan tu moi vao cuoi hang doi
11:    q[0] = i;
12:    return 1;
13:  }
```

c) Loại bỏ một phần tử của hàng đợi

Cài đặt

```
1:  int Remove(Data& i)
2:  {
3:     if (head == -1)
4:        return –1; //Hang doi rong
5:     i = q[head].Data;
6:     head--;
7:     return 1;
8:  }
```

5.3. Hàng đợi được tổ chức theo danh sách liên kết

5.3.1. Tổ chức dữ liệu

```
1:  typedef struct tagItem
2:  {
3:     Data Info;
4:     struct tagItem *Next;
5:     struct tagItem *Prev;
```

```
6:  }item;
7:  item *head, *tail;
```

5.3.2. Các thao tác trên hàng đợi

a) Khởi tạo hàng đợi

Khi khởi tạo, hàng đợi là rỗng, ta cho head và tail có giá trị NULL.

Cài đặt

```
1:  void Initialize()
2:  {
3:     head = NULL;
4:     tail = NULL;
5:  }
```

b) Thêm một phần tử vào hàng đợi

Cài đặt

```
1:  int EnQueue(Data Info)
2:  {
3:     item *i;
4:     //Casp vung nho cho phan tu moi
5:     i = (item*)malloc(sizeof(item));
6:     if  (i == NULL)
7:     return –1;
8:     //Gan thong tin cho phan tu moi
9:     i→Info = Info;
10:    //hang doi chua co phan tu nao
11:    if (tail == NULL)
12:    {
13:    i→Next = NULL;
14:    i→Prev = NULL;
15:    tail = i;
16:    head = i;
17:    }
18:    //Hang doi da co phan tu
19:    else
20:    {
21:    i→Next = tail;
22:    i→Prev = NULL;
23:    tail = i;
24:    }
25:    return 1;
26:  }
```

c) Loại bỏ một phần tử của hàng đợi

Cài đặt

```
1:  int DeQueue(Data& Info)
2:  {
3:    item* i;
4:    if (head == NULL)
5:    return –1;
6:    //Lay thong tin tra ve
7:    Info = head→Info;
8:    //Luu lai dia chi phan tu da lay de xoa di
9:    i = head;
10:   //Di chuyen contro head ve phan tu ngay phia sau
11:   head = i→Prev;
12:   head→Next = NULL;
13:   //Huy phan tu da lay ra
14:   free(i);
15:   return 1;
16: }
```

BÀI TẬP CHƯƠNG 3

Bài tập lý thuyết

1- Phân tích ưu, khuyết điểm của xâu liên kết so với mảng. Tổng quát hoá các trường hợp nên dùng xâu liên kết.

2- Xây dựng một cấu trúc dữ liệu thích hợp để biểu diễn đa thức P(x) có dạng:

$$P(x) = c_1x_{n-1} + c_2x_{n-2} + ... + c_kx_{n-k}$$

Biết rằng:

- Các thao tác xử lý trên đa thức bao gồm:
  - + Thêm một phần tử vào cuối đa thức
  - + in danh sách các phần tử trong đa thức theo:
    - Thứ tự nhập vào
    - ngược với thứ tự nhập vào
  - + huỷ một phần tử bất kỳ trong danh sách
- Số lượng các phần tử không hạn chế
- Chỉ có nhu cầu xử lý đa thức trong bộ nhớ chính.

Giải thích lý do chọn CTDL đã định nghĩa.
Viết chương trình con ước lượng giá trị của đa thứcP(x) khi biết x.
Viết chương trình con rút gọn biểu thức (gộp các phần tử cùng số mũ)

3- Xét đoạn chương trình tạo một xâu đơn gồm 4 phần tử (không quan tâm dữ liệu) sau đây:

```
Dx = NULL; p = Dx;
Dx = new (NODE);
for (i=0; i<4; i++)
{  p = p→ next;
   p = new(NODE);
}
p→next = NULL;
```

Đoạn chương trình có thực hiện được thao tác tạo nêu trên không? tại sao? Nếu không thì có thể sửa lại như thế nào cho đúng?

4- Một ma trận chỉ chứa rất ít phần tử với giá trị có nghĩa (ví dụ: phần tử ≠ 0) được gọi là ma trận thưa.
Ví dụ:

$$\begin{pmatrix} 0 & 0 & 0 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 & 2 & 0 \\ 0 & 0 & 4 & 0 & 0 & 0 \end{pmatrix}$$

Dùng cấu trúc xâu liên kết để tổ chức biểu diễn một ma trận thưa sao cho tiết kiệm nhất (chỉ lưu trữ các phần tử có nghĩa).
Viết chương trình cho phép nhập, xuất ma trận
Viết chương trình con cho phép cộng hai ma trận.

5- Bài toán **Josephus**: có N người đã quyết định tự sát tập thể bằng cách đứng trong vòng tròn và giết người thứ M quanh vòng tròn, thu hẹp hàng ngũ lại khi từng người lần lượt ngả khỏi vòng tròn. Vấn đề là tìm ra thứ tự từng người bị giết.

Ví dụ: N = 9, M = 5 thì thứ tự là 5, 1, 7, 4, 3, 6, 9, 2, 8
Hãy viết chương trình giải quyết bài toán Josephus, xử dụng cấu trúc xâu liên kết.

6- Hãy cho biết nội dung của stack sau mỗi thao tác trong dãy:

EAS*Y**QUE***ST***I*ON

Với một chữ cái tượng trưng cho thao tác thêm chữ cái tương ứng vào stack, dấu * tượng trưng cho thao tác lấy nội dung một phần tử trong stack in lên màn hình.

Hãy cho biết sau khi hoàn tất chuỗi thao tác, những gì xuất hiện trên màn hình?

7- Hãy cho biết nội dung của hàng đợi sau mỗi thao tác trong dãy:

EAS*Y**QUE***ST***I*ON

Với một chữ cái tượng trưng cho thao tác thêm chữ cái tương ứng vào hàng đợi, dấu * tượng trưng cho thao tác lấy nội dung một phần tử trong hàng đợi in lên màn hình.

Hãy cho biết sau khi hoàn tất chuỗi thao tác, những gì xuất hiện trên màn hình?

Bài tập thực hành

8- Cài đặt thuật toán sắp xếp chèn trực tiếp trên xâu kép. Có phát huy ưu thế của thuật toán hơn trên mảng hay không?

9- Cài đặt thuật toán QuickSort theo kiểu không đệ quy.

10- Cài đặt thuật toán MergeSort trên xâu kép.

11- Cài đặt lại chương trình quản lý nhân viên theo bài tập 6 chương 1, nhưng sử dụng cấu trúc dữ liệu xâu liên kết. Biết rằng số nhân viên không hạn chế.

12- Cài đặt chương trình tạo một bảng tính cho phép thực hiện các phép tính +, -, *, / trên các số có tối đa 30 chữ số, có chức năng nhớ (M+, M-, MC, MR).

13- Viết chương trình thực hiện các thao tác trên đa thức

Chương 4: CÂY (TREE)

1. Cây:

1.1. Định nghĩa:

Cây là một tập hợp T các phần tử (gọi là nút của cây) trong đó có 1 nút đặc biệt gọi là gốc, các nút còn lại được chia thành những tập rời nhau T_1, T_2,., T_n theo quan hệ phân cấp trong đó T_1 cũng là một cây. Mỗi nút ở cấp i sẽ quản lý một số nút ở cấp i+1. Quan hệ này người ta gọi là quan hệ cha-con.

1.2. Một số khái niệm cơ bản

- Bậc của một nút : là số con của nút đó.
- Bậc của một cây : là bậc lớn nhất của các nút trong cây (số cây con tối đa của một nút trong cây). Cây có bậc n thì gọi là cây n-phân.
- Nút lá : là nút bậc 0.
- Nút nhánh : là nút có bậc khác 0 và không phải là gốc.
- Mức của một nút :
  + Mức (gốc(T))=1.
  + Gọi $T_1,T_2,T_3,.,T_n$ là cây con của T_0.

Mức (T_1)=Mức(T_2)=Mức(T_3)=…=Mức(T_n)=Mức(T_0)+1.

- Độ dài đường đi từ gốc đến nút x là số nhánh cần đi qua kể từ gốc đến x.
- Độ dài đường đi tổng của cây :

  $P_1 = S\ P_X$ (XT)

  trong đó P_X là độ dài đường đi từ gốc đến X.

  Độ dài đường đi trung bình : $P_1 = P_T / n$ (n : số nút trên cây T).

  Rừng cây: là tập hợp nhiều cây trong đó thứ tự các cây là quan trọng.

1.3. Nhận xét

Trong cấu trúc cây không tồn tại chu trình.

Tổ chức một cấu trúc cây cho phép truy cập nhanh đến các phần tử của nó.

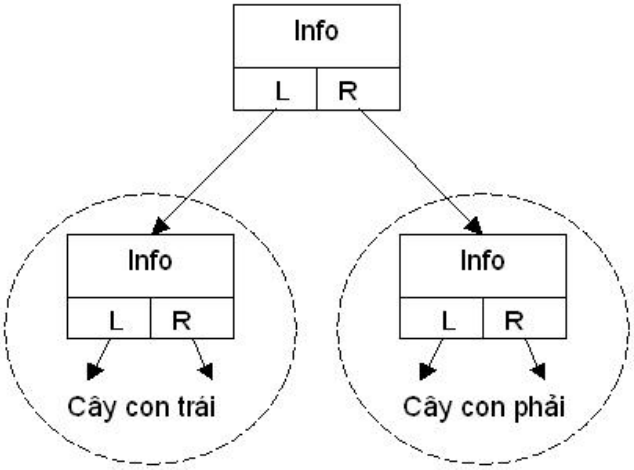
2. Cây nhị phân

2.1. Định nghĩa

Cây nhị phân là cây mà mỗi nút có tối đa 2 cây con.

Trong thực tế thường gặp các cấu trúc có dạng nhị phân. Một cây tổng quát có thể biểu diễn thông qua cây nhị phân.

2.2. Biểu diễn cây nhị phân T



Cây nhị phân thường được biểu diễn theo kiểu cấp phát liên kết: mỗi phần tử ứng với 1 biến động lưu trữ trong Info:

- Thông tin lưu tại nút : Info.
- Địa chỉ nút gốc của cây con trái trong bộ nhớ : L.
- Địa chỉ nút gốc của cây con phải trong bộ nhớ : R.

```
1: typedef   struct    tagNODE
2: {
3:    DATA  Info;
4:    struct  tagNODE   *Left,*Right;
5: }NODE;
6: typedef   NODE   *TREE;
```

Do tính chất mềm dẻo của cách biểu diễn bằng cấp phát liên kết, phương pháp này được dùng chủ yếu trong cây nhị phân. Từ phần này trở đi, khi nói đến cây nhị phân, chúng ta sẽ dùng phương pháp này.

3. Cây nhị phân tìm kiếm

3.1. Định nghĩa

Cây nhị phân tìm kiếm (CNPTK) là cây nhị phân trong đó tại mỗi nút, khoá của tất cả các nút thuộc cây con trái đều nhỏ hơn khoá của nút đang xét và nhỏ hơn khoá của tất cả các nút thuộc cây con phải.

Nhờ ràng buộc về khoá trên CNPTK, việc tìm kiếm trở nên có định hướng. Hơn nữa, do cấu trúc cây, việc tìm kiếm trở nên nhanh đáng kể. Nếu số nút trên cây là N thì chi phí tìm kiếm trung bình chỉ khoảng $\log_2 N$.

Trong thực tế, khi xét đến cây nhị phân chủ yếu người ta xét CNPTK.

3.2. Các thao tác trên cây nhị phân tìm kiếm

3.2.1. Duyệt cây

Có 3 kiểu duyệt chính có thể áp dụng trên cây nhị phân: duyệt theo thứ tự trước (NLR), thứ tự giữa (LNR) và thứ tự sau (LRN). Tên của 3 kiểu duyệt này được đặt dựa trên trình tự của việc thăm nút gốc so với việc thăm 2 cây con.

a) Duyệt theo thứ tự trước (Node-Left-Right)

Kiểu duyệt này trước tiên thăm nút gốc sau đó thăm các nút của cây con trái rồi phải. Thủ tục duyệt có thể trình bày đơn giản như sau:

```
1:  void  NLR(TREE Root)
2:  {
3:    if (Root!=NULL)
4:    {
5:    <Xử lý Root>;  // Xửlý tươngứng theo nhu cầu
6:    NLR(Root→Left);
7:    NLR(Root→Right);
8:    }
9:  }
```

b) Duyệt theo thứ tự giữa (Left-Node-Right)

Kiểu duyệt này trước tiên thăm các nút của cây con trái sau đó thăm nút gốc rồi đến cây con phải. Thủ tục duyệt có thể trình bày đơn giản như sau:

```
1:  void  LNR(TREE Root)
2:  {
3:    if (Root!=NULL)
4:    {
5:    LNR(Root→Left);
6:    <Xử lý Root>;//Xử lý tương ứng theo nhu cầu
7:    LNR(Root→Right);
8:    }
9:  }
```

c) Duyệt theo thứ tự sau (Left-Right-Node)

Kiểu duyệt này trước tiên thăm các nút của cây con trái sau đó thăm đến cây con phải rồi cuối cùng mớithăm nút gốc. Thủ tục duyệt có thể trình bày đơn giản như sau:

```
1:  void  LRN(TREE Root)
2:  {
3:    if (Root!=NULL)
4:    {
5:    LRN(Root→Left);
```

```
6:    LRN(Root→Right);
7:    <Xử lý Root>; //Xửlý tương ứng theo nhu cầu
8:    }
9:  }
```

Lưu ý : ***do tính chất của CNPTK, khi duyệt cây theo thứ tự giữa ta sẽ duyệt các nút theo thứ tự tăng dần của khoá.***

3.2.2. Tìm một phần tử x trong cây

Để tìm một phần tử x trong cây ta chỉ đơn thuần duyệt qua các phần tử cho đến khi tìm thấy x.

```
1:  NODE    *SearchTree(TREE  Root, DATA  x)
2:  {
3:    NODE  *p=Root;
4:    while (p!=NULL)
5:    {
6:    if (x=p→Info)
7:      return p;
8:    else if (x<p→Info)
9:      p=p→Left;
10:   else
11:           p=p→Right;
12:   }
13:   return NULL;
14: }
```

3.2.3. Thêm một phần tử x vào cây

Để thêm một phần tử vào cây ta phải tìm ra vị trí hợp lệ bằng cách gọi đệ quy như sau:

```
1:   void AddTree(TREE   &Root, DaTa  x)
2:  {
3:    if  (Root!=NULL)
4:    {
5:    if (x==Root→Info) //"Báo  X  bị trùng";
6:    else if (x<Root→Info)
7:      AddTree(Root→Left,x);
8:    else
9:      AddTRee(Root→Right,x);
10:   }
11:   else
```

```
12:    {
13:    Root = new NODE;
14:    Root→Info = x;
15:    Root→Left = NULL;
16:    Root→Right = NULL;
17:    }
18:  }
```

3.2.4. Huỷ một phần tử có khoá x

Thuật toán

Bước 1 : Tìm phần tử p có khoá x.

Bước 2 : Có 3 trường hợp :

- p là nút lá : huỷ p.
- p có 1 con : tạo liên kết từ phần tử cha của p đến con của p rồi huỷ p.
- p có 2 con : tìm phần tử thế mạng cho p theo nguyên tắc:
  - "Phần tử thế mạng phải nhất của cây con trái của p (phần tử lớn nhất trong các phần tử nhỏ hơn p)" hay :
  - "Phần tử thế mạng trái nhất của cây con phải của p (phần tử nhỏ nhất trong các phần tử lớn hơn p)."
- Sau đó chép thông tin của phần tử thế mạng vào p và huỷ phần tử thế mạng (do phần tử thế mạng có tối đa một con).

Cài đặt

```
 1:  void DelNode(TREE &Root, DaTa x)
 2:  {
 3:    NODE  *q;
 4:    If (Root==NULL)  return;
 5:    else
 6:    {
 7:    if (x<Root→Info)
 8:      DelNode(Root→Left,x);
 9:    else if (x>Root→Info)
10:      DelNode(Root→Right,x);
11:    else
12:    {
13:      q=Root;
14:      if (q→Right==NULL)
15:           Root=q→Left;
16:      else if (q→Left==NULL)
```

```
17:            Root=q→Right;
18:        else
19:            SearchStandFor(Root→Left,q);
20:        Delete q;
21:    }
22:    }
23: }
```

4. Cây cân bằng

4.1. Định nghĩa

Cây nhị phân tìm kiếm cân bằng là cây mà tại mỗi nút của nó độ cao của cây con trái và của cây con phải chênh lệch không quá một.

4.2. Chỉ số cân bằng của một nút

4.2.1. Định nghĩa

Chỉ số cân bằng của một nút là hiệu của chiều cao cây con phải và cây con trái của nó.

Đối với một cây cân bằng, chỉ số cân bằng (CSCB) của mỗi nút chỉ có thể mang một giá trị sau đây :

CSCB(p) = 0 ⇔ Độ cao cây trái (p) = Độ cao cây phải (p)

CSCB(p) = 1 ⇔ Độ cao cây trái (p) < Độ cao cây phải (p)

CSCB(p) = -1 ⇔ Độ cao cây trái (p) > Độ cao cây phải (p)

Chúng ta ký hiệu như sau :

p→Bal = CSCB(p).

h_L : độ cao cây con trái.

h_R : độ cao cây con phải.

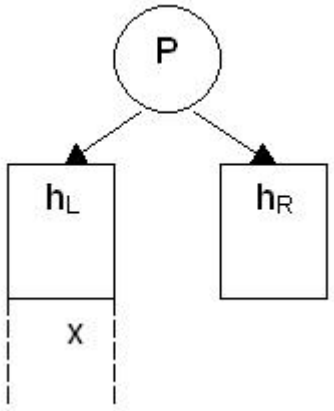
Để khảo sát cây cân bằng ta cần lưu thêm thông tin về chỉ số cân bằng tại mỗi nút. Lúc đó, cây cân bằng có thể được khai báo như sau :

```
1: typedef  struct  tagBAL_NODE
2: {
3:    char   Bal;
4:    DaTa  Info;
5:    struct tagBAL_NODE *Left, *Right;
6: }BAL_NODE;
7: typedef  BAL_NODE  *BALANCE_TREE;
```

Ta nhận thấy trường hợp thêm hay huỷ một phần tử trên cây có thể làm cây tăng hay giảm chiều cao, khi đó phải can bằng lại cây. Việc cân bằng lại một cây sẽ phải thực hiện sao cho ảnh hưởng tối thiểu đến cây nhằm giảm thiểu chi phí cân bằng. Như đã nói ở trên, cây cân

bằng cho phép việc cân bằng lại chỉ xảy ra trong giới hạn cục bộ nên chúng ta có thể thực hiện được mục tiêu vừa nêu.

4.2.2. Thêm vào cây cân bằng



Trước tiên, ta xét các tình huống có thể xảy ra khi ta thêm x vào cây con trái của nút P. Ta thấy có 3 tình huống có thể xảy ra :

- $p\rightarrow Bal = 0$ ($h_L = h_R$) : thêm x vào thì $p\rightarrow Bal=-1$
- $p\rightarrow Bal = 1$ ($h_L < h_R$) : thêm x vào thì $p\rightarrow Bal= 0$
- $p\rightarrow Bal = -1$ ($h_L > h_R$) : thêm x vào thì phải cân bằng lại cây.

Tương tự, nếu thêm một phần tử x vào nhánh phải của P ta cũng có 3 tình huống :

- $p\rightarrow Bal = 0$ ($h_L = h_R$) : thêm x vào thì $p\rightarrow Bal= 1$
- $p\rightarrow Bal = -1$ ($h_L > h_R$) : thêm x vào thì $p\rightarrow Bal= 0$
- $p\rightarrow Bal = 1$ ($h_L < h_R$) : thêm x vào thì phải cân bằng lại cây.

Trong 3 trường hợp trên chỉ có trường hợp3 là cần cân bằng lại cây sau khi thêm do lúc đầu $h_L = h_R + 1$, nếu thêm một phần tử có khả năng làm tăng h_L => $h_L = h_R + 2$.

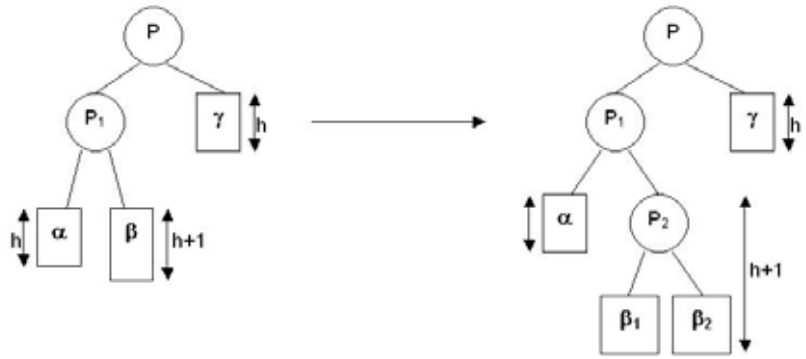
Xét trường hợp phải cân bằng lại do cây bị lệch về nhánh trái ($h_L>h_R$). Ta nhận thấy có 2 trường hợp :

Cây con trái p_L của p có CSCB $p_L\rightarrow Bal = -1$:

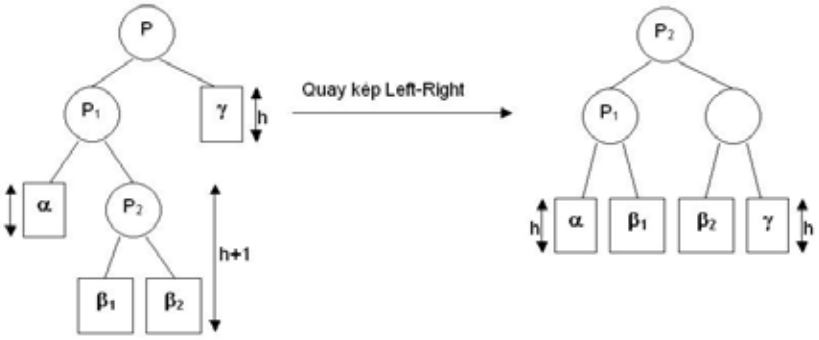
Như trong hình vẽ ta thấy, để cân bằng lại ta thực hiện phép "quay trái LL". Sau khi quay xong, $p_L\rightarrow Bal=0$ và $p\rightarrow Bal=0$. Cây con trái p_L của p có CSCB $p_L\rightarrow Bal= 1$:

Trường hợp này phức tạp hơn. Ta không thể quay trái LL như trường hợp trên. Để cân bằng lại phải tiến hành "quay kép Left-Right" như trong hình vẽ bên dưới.

Trước tiên ta phân tích cây p chi tiết thêm một bậc. Ta gọi p_2 là cây con phải của p_L. Cây p lúc đó sẽ có hình dáng như sau:



Kết quả của phép quay LR được trình bày trong hình dưới đây :



Sau khi quay $p_2 \rightarrow$Bal=0.

Nếu trước khi quay $p_2 \rightarrow$Bal =-1 thì sau khi quay $p_L \rightarrow$Bal=0; $p \rightarrow$Bal = 1.

Nếu trước khi quay $p_2 \rightarrow$Bal = 1 thì sau khi quay $p_L \rightarrow$Bal = -1; $p \rightarrow$Bal = 0.

Tương tự, nếu lúc đầu $h_R = h_L + 1$, $p \rightarrow$Bal=1 và thêm một phần tử ở bên phải. Giả sử việc thêm vào làm cây tăng trưởng ($h_R = h_L + 2$) => cân bằng lại cây. Gọi p1 là cây con phải của p. Ta có các tình huống sau :

$p_1 \rightarrow$Bal = 1 : ta phải quay đơn Right-Right.

Sau khi quay $p_1 \rightarrow$Bal=0; $p \rightarrow$Bal =0.

$p_1 \rightarrow$Bal = 1 : ta phải quay kép Right-Left.

Nếu p_2 là cây con trái của p_1, sau khi quay $p_2 \rightarrow$Bal = 0. Đối với p và p_1 ta có 2 trường hợp xảy ra tương tự như đối với quay kép Left-Right.

Thuật toán

Bước 1 : Đi theo đường tìm kiếm để xác định phần tử cần thêm vào cây.

Bước 2 : Nếu phần tử cần thêm chưa tồn tại trên cây thì :

- Thêm phần tử đó vào cây.
- Xácđịnh lại hệ số cân bằng của phần tử mới thêm.

Bước 3 : Lần ngượctheo đườngtìm kiếm và kiểm tra hệ số cân bằng tại mỗi nút, nếu thấy mất cân bằng thì phải cân bằng lại.

Nhận xét

Trong thực tế việc cân bằng lại khi thêm một nút vào cây chỉ thựchiện ở vị trí cân bằng chứ không cần phải làm ở mức cha của nó.

```
h = FALSE ;//là biến toàn cục.
void Add_BalanceTree(TREE &p, DaTa x)
// Thêm x vào cây p, h cho biết cây tăng trưởng không?
// nếu h=FALSE : cây cân bằng
{
  NODE *p1, *p2;
  if (p==NULL)
  {
    p = new NODE;
    p→Info = x;
    p→Left = NULL;
    p→Right = NULL;
    p→Bal = 0;
    h = TRUE;
  }
  else if (x<p→Info)
  {
    Add_BalanceTree(p→Left,x);
    if (h)
        switch (p→Bal)
        {
            case 0 :
            {
                p→Bal = -1;
                break;
            }
            case 1:
            {
                p→Bal:=0;
                break;
            }
            case -1:
            {
                p1=p→p→Left;
                if (p1→Bal==-1)//quay đơn Left-Left
```

```
36:                 {
37:                     p→Left=p1→Right;
38:                     p1→Right=p;
39:                     p→bal=0;
40:                     p1→bal=0;
41:                     p=p1;
42:                 }
43:                 else
44:                     //<quay kép Left-Right>
45:                 h=false;
46:             }
47:         }
48:     }
49:     else if (x>p->Info)
50:     {
51:         Add_BalanceTree(p->Right,x);
52:         if (h)
53:             //<Cân bằng lại tương tự>
54:     }
55:     else  // x= p->Info, đã có x trong cây
56:         h=false;
57: }
```

4.2.3. Huỷ trên cây cân bằng

Các trường hợp loại bỏ một phần tử trên cây cân bằng vẫn được giải quyết như khi loại bỏ trên cây nhị phân tìm kiếm.

Việc loại bỏ một phần tử có khả năng là giảm độ cao của cây dẫn đến việc phải cân bằng lại do cây mất cân bằng. Việc cân bằng lại có thể lan truyền ngược lên những phần tử phía trên.

Các trường hợp cân bằng lại giải quyết giống thêm vào.

Thuật toán

Nếu p= NULL thì " Không có phần tử để huỷ"

Else Nếu x<p→Info thì

Tiến hành huỷ cây con trái

Cân bằng lại cho p nếu cây con trái giảm đọ cao (Balance-Left)

Else Nếu x>p→Info thì

Thực hiện huỷ cây con phải

Cân bằng lại cho p nếu cây con phải giảm độ cao (Balance-Right)

Else {x=p→Info}

Huỷ p giống thuật toàn trên cây nhị phân tìm kiếm.

Cài đặt

```
// h = FALSE là biến toàn cục.
1:  void Del_BalanceTree(TREE &p)
2:  {
3:    NODE *p;
4:    if (p==NULL)//Không có trên cây cân bằng
5:    h=false;
6:    if (p→Info>x)
7:    {
8:    Del_BalanceTree(p→Left,x);
9:    if (h)
10:     Balance_Left(p);
11:   }
12:   else if (x>p→info)
13:   {
14:   Del_BalanceTree(p→Right,x);
15:   if (h)
16:     Balance_Right(p);
17:   }
18:   else
19:   {
20:   q=p;
21:   if (q→Right==NULL)
22:   {
23:     p=p→Left;
24:     h=true;
25:   }
26:   else if (q->Left==NULL)
27:   {
28:     p=p→Right;
29:     h=true;
30:   }
31:   else
32:   {
33:     Search_StandFor(p→Left,&q);
34:     if (h)
35:         Balance_Left(p);
```

```
36:    }
37:    delete q;
38:    }
39:  }
```

Sau đây là thủ tục cân bằng cây con trái và cây con phải:

```
 1:  void Balance_Right(TREE &p)
 2:  {
 3:    NODE *p1, *p2;
 4:    int b1,b2;
 5:    switch (p→Bal)
 6:    {
 7:      case 1:
 8:           p→Bal =0;
 9:           break;
10:      case 0 :
11:           p→Bal=-1;
12:           break;
      case -1 :
13:      {
14:           p1=p→Left;
15:           b1=p1→Bal;
16:           if (b1<=0)
17:           {
18:                //Quay don LL
19:                p→Left = p1→Right;
20:                p1→Right=p;
21:                if (b1=0)
22:                {
23:                     p→Bal=-1;
24:                     p1→Bal=-1;
25:                     h=false;
26:                }
27:                else
28:                {
29:                     p→Bal=0;
30:                     p1→Bal=0;
31:                }
32:                     p=p1;
33:           }
34:           else
```

```
35:          {
36:              p2=p1→Right;
37:              b2=p2→Bal;
38:              //Quay kep RL
39:              p→Left=p2→Right;
40:              p2→Right=p;
41:              p1→Right=p2→Left;
42:              p2→Left=p1;
43:              if (b2==0)
44:              {
45:                  p1→Bal=0;
46:                  p→Bal=0;
47:                  p2→Bal=0;
48:              }
49:              else if (b2==1)
50:              {
51:                  p1→Bal=-1;
52:                  p→Bal=0;
53:                  p2→Bal=0;
54:              }
55:              else
56:              {
57:                  p1→Bal=0;
58:                  p→Bal=0;
59:                  p2→Bal=0;
60:              }
61:              p=p2;
62:              h=true;
63:          }
64:      }
65:   }
66: }
```

Hàm Balance_Left xây dựng hoàn toàn tương tự.

```
1:  void Search_StandFor(TREE &R, TREE &q);
2:  {
3:    if (R→Right!=NULL)
4:    {
5:    Search_StandFor(R →Right,q);
6:    if (h)
```

```
 7:       Balance_Rigth;
 8:    }
 9:    else
10:    {
11:    q→Infor=R→Info;
12:    q=R;
13:    R=R→Left;
14:    h=true;
15:    }
16: }
```

5. Cây nhiều nhánh

5.1. Định nghĩa

Ta nhận thấy cây cân bằng đòi hỏi một tiêu chuẩn cân bằng lại trong quá trình cây bị biến đổi, mà việc cân bằng bao gồm nhiều thao tác phức tạp. Một tiêu chuẩn rất hợp lý được R.Bayer đưa ra vào năm 1970 là:

Mọi trang (trừ trang gốc) chứa ít nhất n nút và nhiều nhất 2*n nút với n là hằng số cho trước.

Do đó cây có N phần tử và mỗi trang có tối đa 2*n nút thì trường hợp xấu nhất đòi hỏi $\log_n N$ truy xuất trang và việc truy xuất trang chi phối toàn bộ công sức tìm kiếm. Hơn nữa hệ số sử dụng bộ nhớ ít nhất là 50% vì mỗi trang chứa ít nhất n phần tử.

Nhờ các ưu điểm này mà ta có thể thực hiện dễ dàng các phép tìm kiếm, thêm vào và loại bỏ.

5.2. B-cây

5.2.1. Các khái niệm

Cấu trúc dữ liệu B-cây cấp n có các đặc tính sau:

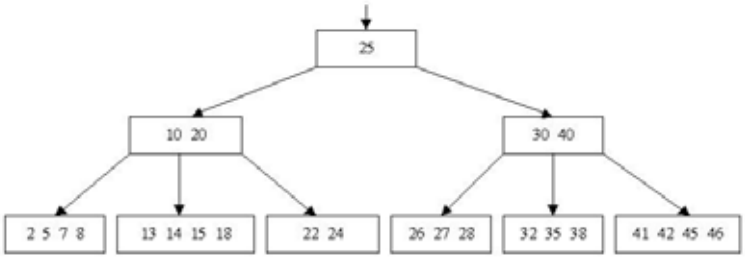
1. Mỗi trang có tối đa 2*n phần tử (khóa).

2. Mỗi trang, ngoại trừ trang gốc, có ít nhất n phần tử.

3. Mỗi trang hoặc là trang lá (không có con) hoặc có m + 1 trang con, với m là số khóa của trang này.

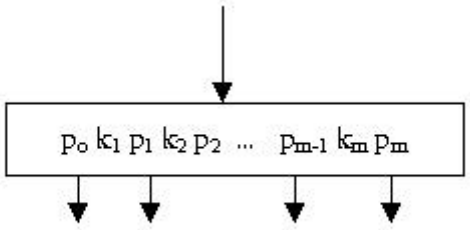
4. Tất cả các trang lá phải có cùng mức.

Ví dụ:



Đây là một B-cây cấp 2 có ba mức. Tất cả các trang chứa 2,3,4 phần tử; ngoại trừ trang gốc được phép chỉ chứa một phần tử. Tất cả các trang lá xuất hiện cùng một mức.

Các khóa xuất hiện tăng dần từ trái qua phải nếu Bước-cây được gom lại trong cùng một mức bằng cách xen các nút con giữa các khóa của nút cha của chúng. Sự sắp xếp này biểu diễn sự mở rộng tự nhiên của việc tổ chức các cây nhị phân và nó xác định phương pháp tìm kiếm một phần tử với khóa cho trước.



Xét một trang sau đây và ta cần tìm một phần tử có khóa là x cho trước. Giả sử rằng trang đã được đưa vào trong bộ nhớ. Ta có thể sử dụng các phương pháp tìm kiếm đã trình bày ở trên: nếu m đủ lớn thì ta dùng phương pháp tìm kiếm nhị phân, nếu m nhỏ thì ta dùng phương pháp tìm kiếm tuần tự thông thường. Lưu ý rằng thời gian tìm kiếm trong bộ nhớ chính có thể không đáng kể. Như vậy với thời gian để đọc một trang từ bộ nhớ ngoài bộ nhớ chính.

Nếu không tìm thấy thì ta có các trường hợp sau:

1. $k_i < x < k_{i+1}$ với $1 <= i < m$. Ta tiếp tục tìm kiếm trên trang p_i^

2. khoá $k_m < x$. a tiếp tục tìm kiếm trên trang phép pm ^

3. $x < k_1$. Ta tiếp tục tìm kiếm trên trang p_0 ^

Nếu trong trường hợp chỉ điểm chỉ đến null, nghĩa là không có trang con thì khóa x không có trên cây và việc tìm kiếm kết thúc.

Như vậy việc tìm kiếm trên B-cây rất đơn giản.

Ta khai báo dữ liệu của B-cây như sau:

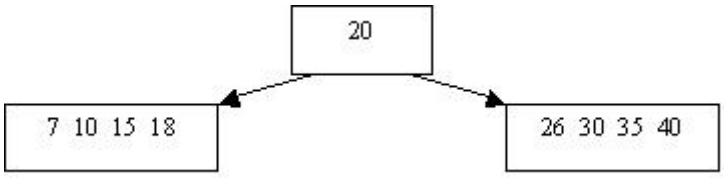
```
 1:  #define n 2
 2:  typedef struct
 3:   {
 4:    int key;
 5:    int count;
 6:  }item;
 7:  typedef struct tagPage
 8:   {
 9:    int m;
10:    struct tagPage * p0;
11:    item e[n];
12:   }page;
```

5.2.2. Tìm kiếm và thêm vào trên B-cây

- Nếu một phần tử được thêm vào một trang chưa đầy ($m < 2 * n$) thì quá trình thêm vào chỉ xảy ra trên trang này.

- Nếu một phần tử được thêm vào một trang đã đầy (m = 2 * n) thì quá trình thêm vào sẽ ảnh hưởng đến cây và có thể phải cấp phát trang mới.

Ví dụ: xét B-cây sau đây



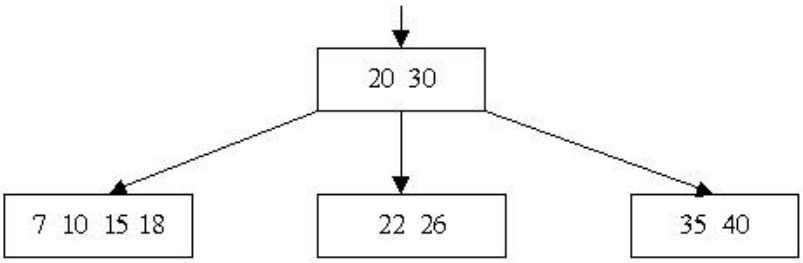
Và ta thêm khóa 22 vào cây.

Các bước thực hiện như sau:

1. Trang A không có khóa 22 và ta tiếp tục tìm kiếm trên trang C. Trang C không có khóa 22. Ta không thể thêm khóa 22 vào trang C này được vì trang C đã đầy (mỗi trang của B-cây cấp 2 chỉ chứa tối đa bốn phần tử).

2. Trang C được tách thành hai trang và ta cần cấp phát thêm trang mới là D.

3. m +1 khóa được phân phối đều nhau vào trang C và trang D, khóa ở chính giữa được chuyển lên trang cha A.



Cây sau khi bị biến đổi vẫn thỏa mãn các đặc tính của một B-cây.

Các trang tách chỉ chứa đúng n phần tử. Trường hợp khóa được chuyển lên trang cha có thể làm cho trang cha bị tràn. Như thế gây ra việc tách trang lan truyền và có thể lan truyền tới trang gốc(trang gốc bị tách làm hai trang). Đây chính là cách duy nhất mà B-cây có thể tăng chiều cao: nó lớn lên từ lá đến gốc.

Vì quá trình tách trang lan truyền dọc ngược theo đường tìm kiếm nên ta sử dụng giải thuật đệ qui và quá trình này cũng tương tự với quá trình thêm vào cây cân bằng.

Giải thuật tìm kiếm và thêm vào được viết thành thủ tục Search. Thủ tục này tương tự với thủ tục thêm vào cây cân bằng quyết định rẽ nhánh không phải là việc chon lựa nhị phân (đến nút con bên trái hoặc đến nút con bên phải) mà là việc tìm kiếm nhị phân trên trang (mảng e). Trong giải thuật, ta dùng biến h kiểu boolean cho biết cây con đã lớn lên. Nếu h có giá trị là trục thì tham số biến thứ hai u là phần tử được đưa lên trang cha.

x - khoa dang tim

a - trang hien tai dang tim khoa x

```
1:  void search(int x, page* a, int& h, item& u)
```

```
 2:  {
 3:    if (a = NULL)
 4:    {
 5:    //x khong co tren cay
 6:    "gan x cho u, cho h la true,de chi ra rang
 7:    mot phan tu u duoc chuyen len tren cay"
 8:    }
 9:    else
10:    {
11:    //tim kiem x trong trang a
12:    if "tim thay"
13:    // tang so lan xuat hien khoa x len 1'
14:    else
15:    {
16:      search (nut ,nutcon ,h,u );
17:      if (h) //mot phan tu u duoc chuyen len
18:              if " so phan tu cua trang  a < 2n"  then
19:              " xen u vao trang a va cho h la false »
20:          else
21:              //tach trang va chuyen phan tu giua len
22:    }
23:    }
24:  }
```

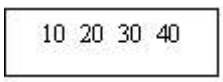
Nếu tham số biến h là true sau khi gọi thủ tục Search trong chương trình chính thì điều này có nghĩa là trang gốc bị tách trang gốc phải được lập trình riêng biệt, gồm cấp phát trang mới (gốc) và thêm vào một phần tử cho bởi tham số biến u. Do đó trang gốc mới chỉ có một phần tử.

Ví dụ: Ta tạo một B-cây cấp 2 từ dãy các khóa sau đây:

20; 40 10 30 15; 35 7 26 18 22; 5; 42 13 46 27 8 32; 38 24 45 25;

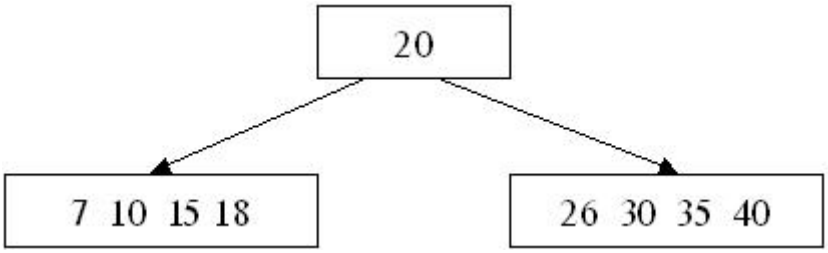
Các dấu chấm phẩy (;) chỉ ra các vị trí "đột biến" mỗi khi có sự cấp phát trang.

Cây sau khi thêm vào khóa 30 như sau:



Khi thêm vào khóa 15 thì trang này bị đầy (một trang B-cây cấp hai chứa tối đa bốn phần tử). Dãy các khóa là 10 15 20 30 40 nên trang này bị tách ra thành hai trang: một trang chứa khóa (10 15) và một trang chứa khóa (30 40), và cấp phát thêm trang mới chứ khóa 20 (trang cha).

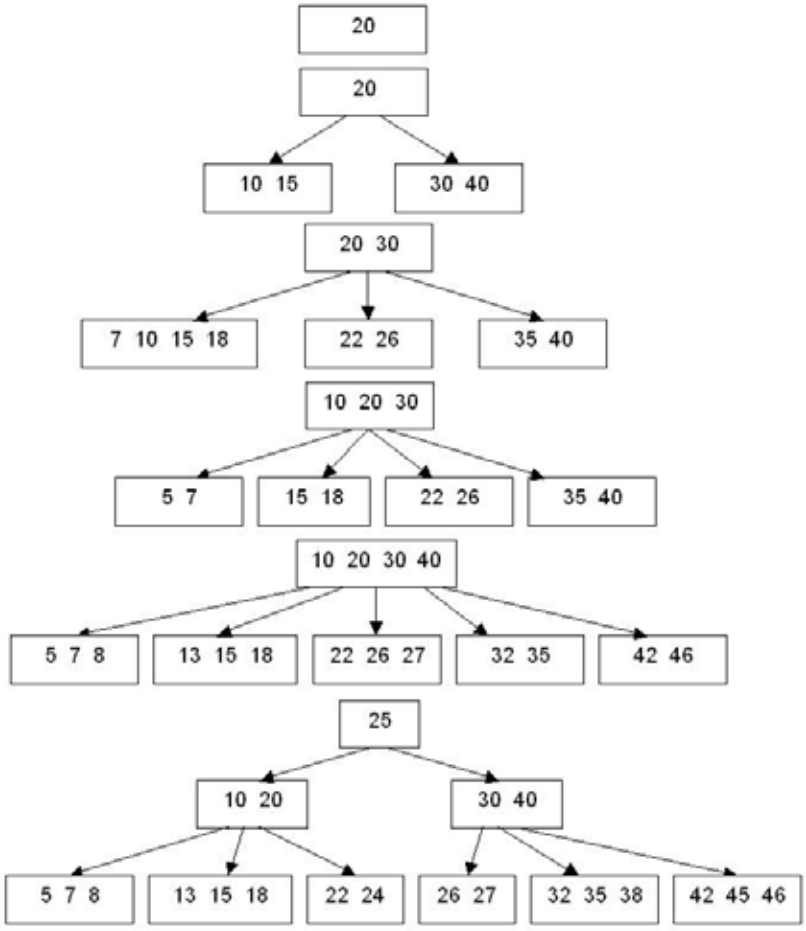
Cây sau khi thêm vào khóa 18 như sau:



Khi thêm khóa 22 vào trang (26 30 35 40) thì làm cho trang này bị đầy và phải tách thành hai trang: một trang chứa khóa (22 26) và một trang chứa khóa (35 40), khóa 30 được đưa lên trang cha và trở thành trang (20 30).

Sự thêm vào của khóa cuối cùng gây ra hai lần tách trang và cấp phát ba trang mới. Khi thêm khóa 25 vào trang (22 24 26 27) làm trang này bị tách thành hai trang và khóa 25 được đưa lên trang cha (10 20 30 40), nhưng trang này bị tràn và bị tách thành hai trang với khóa 25 được đưa vào trang gốc mới.

Hình 5.37 cho ta thấy kết quả của giải thuật tìm kiếm và thêm vào để xây dựng B-cây cấp hai với dãy các khóa đã cho ở trên:



Ghi chú:

Phát biểu ***with*** trong giải thuật trên có một ý nghĩa đặc biệt. Tên của các thành phần trong trang được tự động hiểu là của trang a^. Trong thực tế, nếu các trang được cấp phát trên bộ nhớ ngoài - là điều cần thiết trong hệ cơ sở dữ liệu lớn - thì phát biểu with được diễn dịch thêm là chuyển trang được chỉ định vào trong bộ nhớ chính. Vì mỗi lần thủ tục Search gọi sẽ dẫn đến việc cấp phát một trang trong

Bộ nhớ chính, nên cần nhiều nhất là $k = \log_n N$ lần gọi đệ qui. Do đó, nếu cây có N phần tử thì bộ nhớ chính phải đủ chứa khóa trang. Đây là hệ số giới hạn cho trang có kích thước 2*n. Thực ra ta cần nhiều hơn khóa trang bởi vì khi thêm vào có thể gây ra sự tách trang. Một hệ quả là tốt nhất nên để trang gốc thường xuyên trong

Bộ nhớ chính bởi vì mỗi yêu cầu tìm kiếm đều phải bắt đầu từ gốc.

Một ưu điểm khác của B-cây là sự thích hợp và tiết kiệm của nó trong trường hợp cập nhật thuần túy tuần tự của toàn bộ cơ sở dữ liệu. Mỗi trang được đưa vào bộ nhớ chính đúng một lần.

5.2.3. Tìm kiếm và loại bỏ trên B-cây

Về nguyên tắc, việc loại bỏ các phần tử của B-cây là hoàn toàn đơn giản nhưng phức tạp trong chi tiết. Ta có thể phân biệt hai tình huống khác nhau:

1. Phần tử bị loại bỏ ở trang lá: trong trường hợp này việc loại bỏ thật rõ ràng và đơn giản.

2. Phần tử bị loại bỏ không ở trang lá: nó phải được thay thế bởi một trong hai phần tử kề của nó nằm ở các trang lá và có thể bị loại bỏ dễ dàng.

Trong trường hợp 2, việc tìm khóa kề tương tự như việc tìm khóa trong phép loại bỏ trên cây nhị phân. Ta đi xuống dọc theo các chỉ điểm cực phải để đến trang lá P, thay thế phần tử bị loại bỏ bởi phần tử cuối cùng của trang P này và sau đó giảm kích thước của trang P đi 1.

Trong mọi trường hợp, việc giảm kích thước của trang phải kèm theo việc kiểm tra số phần tử m trên trang bị giảm. Nếu $m < n$ thì đặc tính thứ hai của cây B-cây bị vi phạm. Ta cần phải thêm một vài thao tác; tham số biến heap kiểu Boolean chỉ ra điều kiện cạn này (underflow condition).

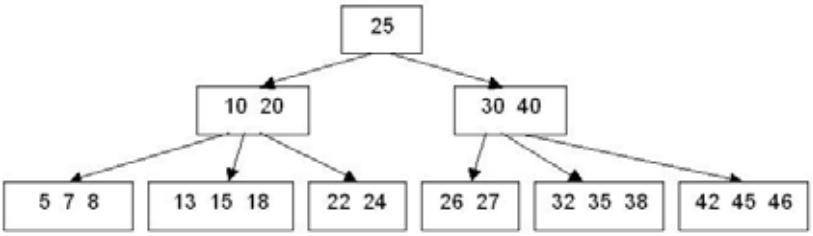
Đối với trang bị cạn, ta chỉ có cách giải quyết là mượn hay "nối" một phần tử từ một trong các trang lân cận. Vì việc này đòi hỏi phải chuyển trang Q (là trang anh em bên trái hoặc bên phải của trang P). vào trong bộ nhớ chính - một thao tác tương đối tốn kém - ta muốn giải quyết tốt nhất tình huống bất lợi đó và nối nhiều phần tử với nhau cùng một lúc. Chiến lược thông thường là phân bố đều các phần tử trên cả hai trang P và Q. Điều này được gọi là làm cân bằng (balancing).

Tất nhiên có thể xảy ra trường hợp là không có phần tử nào để nối sang bởi vì Q vừa đạt kích thước tối thiểu n. Trong trường hợp này, tổng số phần tử trên các trang P và Q là 2 * n -1; ta có thể trộn (merge) hai trang thành một trang cùng với việc thêm vào phần tử giữa của trang cha của P và Q, sau đó hủy bỏ trang Q. Đây chính là quá trình ngược của sự tách trang.

Một lần nữa, việc loại bỏ một phần tử của trang cha có thể làm cho kích thước của trang này nhỏ hơn giới hạn n (trang cha bị cạn). Do đó ta lại cần phải có thao tác trộn trang ở mức thấp hơn (cân bằng hay trộn). Việc trộn trang có thể lan truyền đến trang gốc. Nếu kích thước của trang gốc bị giảm xuống 0 thì ta loại bỏ trang gốc này và do đó làm giảm chiều cao của cây. Đây chính là cách duy nhất để cây B-cây giảm chiều cao.

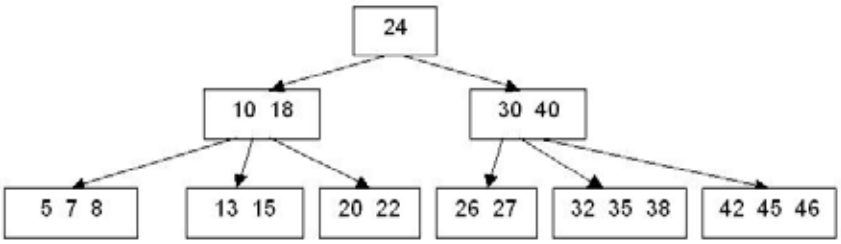
Ví dụ: Xét B-cây được cho ở hình 5.38 và ta lần lượt loại bỏ các khóa sau đây:

25 45 24; 38 32; 8 27 46 13 42; 5 22 18 26; 7 35 15;

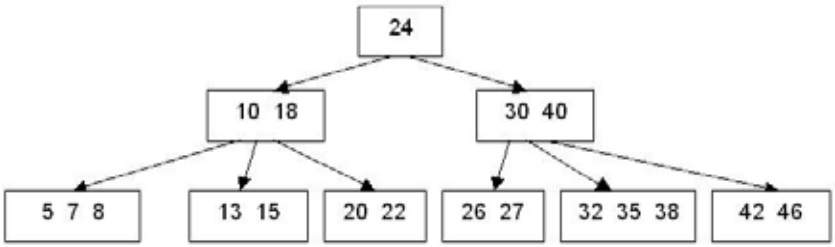


Các dấu chấm phẩy (;) cũng lại chỉ ra các vị trí "đột biến" nơi mà trang bị loại bỏ. Giải thuật loại bỏ của B-cây cũng tương tự với giải thuật loại bỏ trên cây cân bằng.

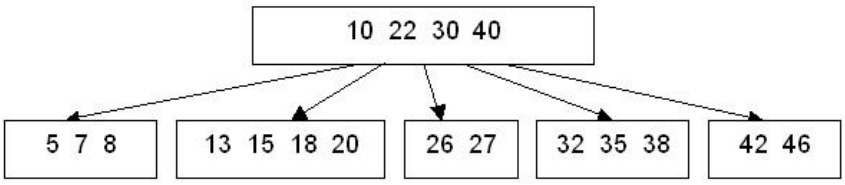
Khi loại bỏ khóa 25, ta đem khóa 24 của trang F (phần tử cuối cùng của trang cực phải ở cây con bên trái của trang A) vào thay thế khóa 25. Khi đó trang F chỉ chứa một khóa 22 (underflow), ta phải đem khóa 20 của trang B vào trang F và đem khóa 18 của trang E vào trang B.



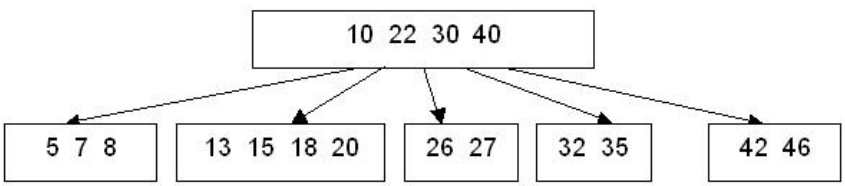
Cây sau khi loại bỏ khóa 45 như sau:



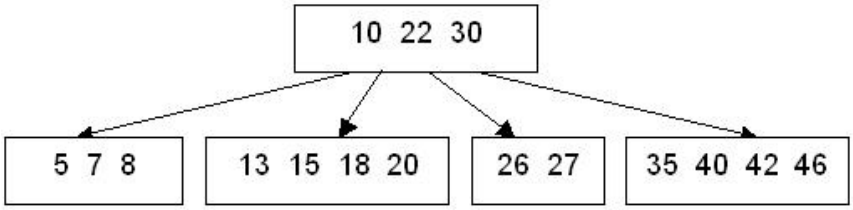
Khi đã loại bỏ khóa 24, ta đem khóa 22 của trang F vào thay thế khóa 24. Khi đó trang F chỉ chứa một khóa 20 (underfow), ta phải nhập trang F vào trang E, đồng thời đem khóa 18 của trang B vào trang E, sau đó hủy bỏ trang F. Như vậy trang E chứa các khóa (13 15 18 20). Nhưng khi đó trang B chỉ chứa một khóa 10 (underflow) nên ta phải nhập trang C vào trang B đồng thời đưa khóa 25 vào trang B, sau đó hủy bỏ trang C và A. Trang B chứa các khóa (10 22 30 40)



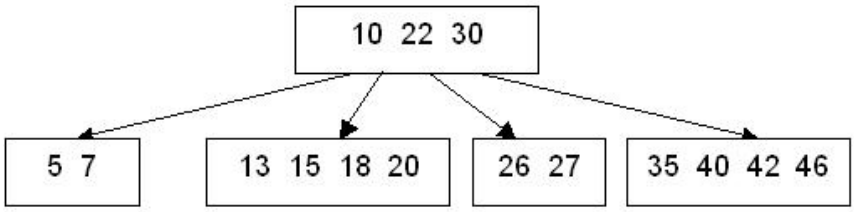
Cây sau khi loại bỏ khóa 38:



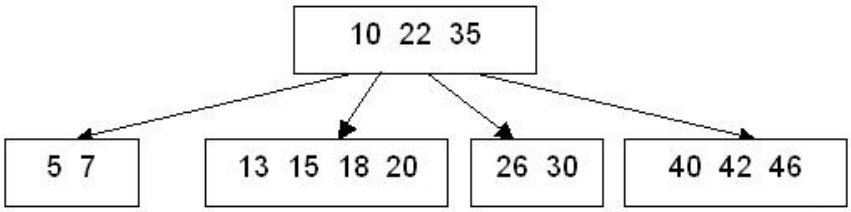
Khi loại bỏ khóa 32, trang H chỉ chứa một khóa 35 (underflow), ta phải nhập trang I vào trang H đồng thời đem khóa 40 của trang B vào trang H, sau đó hủy bỏ trang I. Trang H chứa các khóa (35 40 42 46).



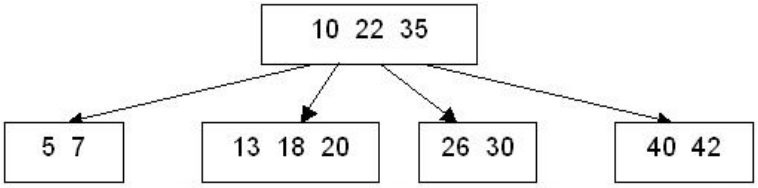
Cây sau khi loại bỏ khóa 8:



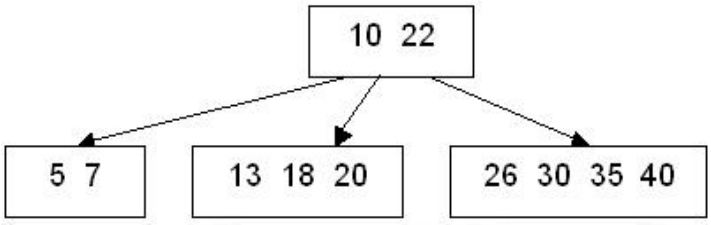
Khi loại bỏ khóa 27, ta đem khóa 30 của trang B vào trang G và khóa 35 của trang B vào trang G, sau đó hủy bỏ trang H.



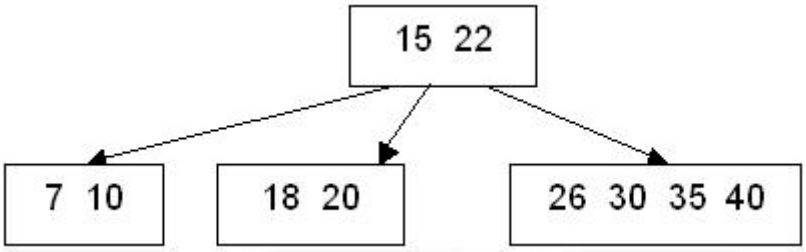
Cây sau khi loại bỏ khoá 13.



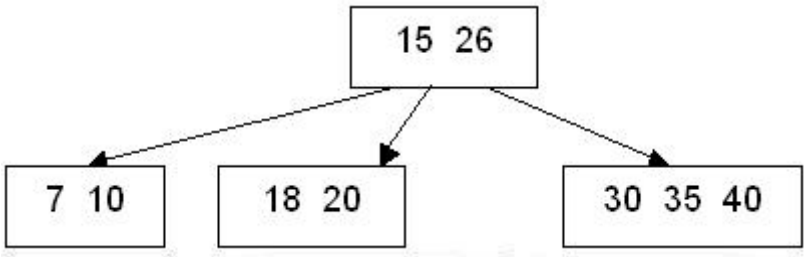
Khi loại bỏ khoá 42, ta nhập trang H vào trang G đồng thời đem khoá 35 của trang B vào trang G, sau đó huỷ bỏ trang H.



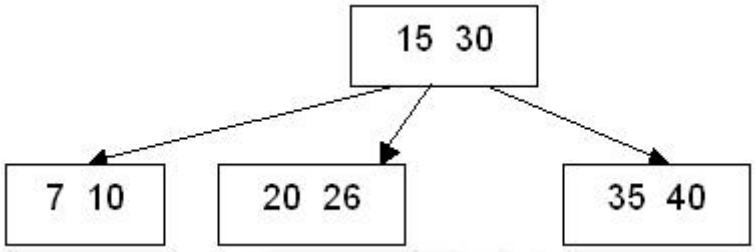
Khi loại bỏ khóa 5, ta đem khóa 10 của trang B vào trang D và khóa 15 của trang E vào trang B.



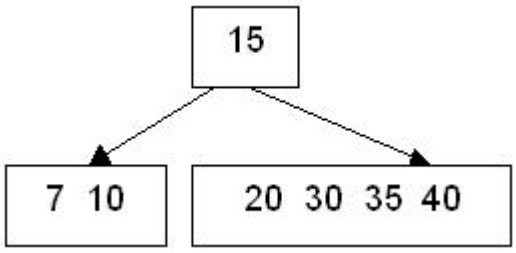
Khi loại bỏ khóa 22, ta đem khóa 20 của trang E vào trang B. khi đó trang chỉ chứa khóa 18 (underflow) nên ta phải đem khóa 20 trở lại vào trang B.



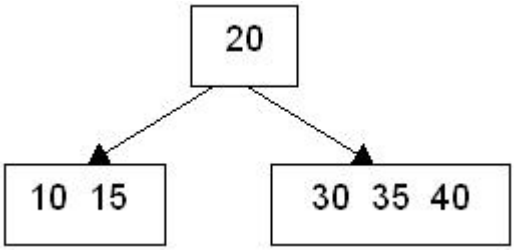
Khi loại bỏ khóa 18, ta đem khóa 26 của trang B vào trang E và khóa 30 của trang G vào trang B.



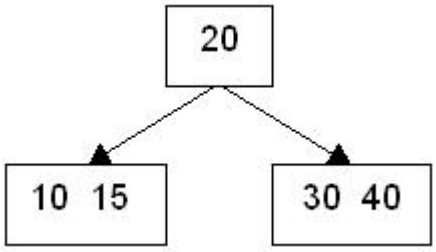
Khi loại bỏ khóa 26, ta nhập trang G vào trang E, đồng thời đem khóa 30 của trang B vào trang E, sau đó hủy bỏ trang G.



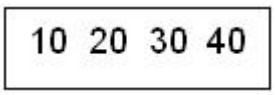
Khi loại bỏ khóa 7, ta đem khóa 15 của trang B vào trang D và khóa 20 của trang E vào trang B.



Cây sau khi loại bỏ khóa 35:



Khi loại bỏ khóa 15, ta nhập trang E vào trang D, đồng thời đem khóa 20 của trang B vào trang D, sau đó hủy bỏ trang E và trang B.



Giải thuật tìm kiếm và loại bỏ được viết thành như thủ tục delete như sau:

```
1:  void delete (int x , page* a, int& h)
2:  {
3:    /*timkiem va loai bo hoa x tren b_cay ; neu trang a co  it hon n phan tu thi dieu chinh
      voi trang ke hoat    tron lai ; h : " trang a co it ghon n phan tu */
4:    int i, k, t, r;
5:    page* q;
6:    if (a == NULL)
7:      {
8:      writeln ( 'khoa khong co tren cay');
9:      h:=false
10:   }
11:   else
12:   {
13:     //tim kiem nhi phan
14:     t: = 2 ; r:= m ;
15:     do
16:     {
```

```
17:          k:=(t +r) div 2 ;
18:          if x  <= e[k] .key then r:=k -1 ;
19:          if x> = e[k] .key then t:= k+1
20:      }
21:      while (t <= r);
22:      if (r == 0)
23:          q = p0
24:      else
25:          q:= e[r].p;
26:      if (t-r >1)
27:      {
28:          //tim thay , loai bo e[k]
29:          if q= nil then
30:          {
31:              //a la trang la}
32:              m:= m -1 ;
33:              h:= m< n ;
34:              for i:= k to m do e[i] := e[i+1]
35:          }
36:          else
37:          {
38:              Del( q,h);
39:              if (h)
40:                  underflow (a,q,r,h);
41:          }
42:      }
43:      else
44:      {
45:          Delete (x,q,h);
46:          if (h)
47:              underflow (a.q.r.h ); //cài đặt phía dưới
48:      }
49:    }
50:  }
```

Hàm underflow:

```
1:  void Underflow (page* c, page* a, int s, int& h)
2:  //a = trang co it hon n phan tu , c= trang cha
3:  {
4:    page* b;
```

```
int i, k, mb, mc;
mc = c→m; //h= true,a→m = n – 1
if (s < mc) //b: trang ben phai cua trang a
{
  s ++;
  b = c→e[ s].p;
  mb = b→m;
  k = (mb –n +1) / 2 ;
  //k= so phan tu se de lai tren trang ke b
  a→e[n] = c→e[s] ;
  a→e[n] .p = b→p0;
  if (k>0) //chuyen k phan tu tu trang a
  {
      for (i = 0; i < k-1; i++)
          a→e[i+n] = b→e[i];
      c→e[s] = b→e[k];
      c→e[s].p = b;
      b→p0 = b→e[k].p;
      mb = mb- k ;
      for (i = 0; i <  mb; i++)
          b→e[i] = b→e[i+k];
      b→m = mb;
      a→m = n -1 +k;
      h = false;
  }
  else //tron trang a voi trang b
  {
      for (i = 0; i <  n; i++)
          a→e[i+n] = b→e[i] ;
      for (i =s; i < mc – 1; i++)
          c→e[i] =c→e[i+1];
      a →m = nn;
      c→m = mc -1 ;
      delete b;
  }
}
else //b = trang ben trai cua trang a
{
  if (s == 1)
      b = c→p0;
  else
```

```
46:          b = c→e[s-1] .p;
47:      mb = b→m +1 ;
48:      k = (mb –n)/2 ;
49:      if (k >0)//chuyen k phan tu tu trang b vao trang a
50:      {
51:          for (i = n -1; i >= 0; i--)
52:              a→e[i=k] =a→e[i];
53:          a→e[k] = c→e[s] ;
54:          a→e[k] .p = a→p0;
55:          mb -= k ;
56:          for (i = k -1; k >= 0; k--)
57:              a→e[i] = b→e[i +mb];
58:          a→p0 = b→e[mb] .p;
59:          c→e[s] = b→e[mb] ;
60:          c→e[s].p = a ;
61:          b→m = mb -1 ;
62:          a→m = n -1 + k ;
63:          h = false;
64:      }
65:      else// tron trang a voi trang b
66:      {
67:          b→e[mb] = c→e[s] ;
68:          b→e[mb] .p = a→p0;
69:          for (i = 0; i < n -1; i++)
70:              b→e[i + mb] = a→e[i];
71:              b→m = nn;
72:          c→m = mc -1 ;
73:          delete a;
74:      }
75:   }
76: }
```

```
 1: void  Del  ( page* p,int &h)
 2: {
 3:   page* q;
 4:   q = p→e[m].p;
 5:   if (q != NULL)
 6:   {
 7:     del (q,h) ;
 8:     if h
 9:          Underflow ( p,q ,m,h);
10:   }
```

```
11:     else
12:     {
13:       p→e[m].p = a→e[k].p;
14:       a→e[k] := p→e[m];
15:       m = m-1 ;
16:       h = m <  n;
17:     }
18:   }
```

6. B-cây nhị phân (Binary B-câytree)

6.1. Giới thiệu

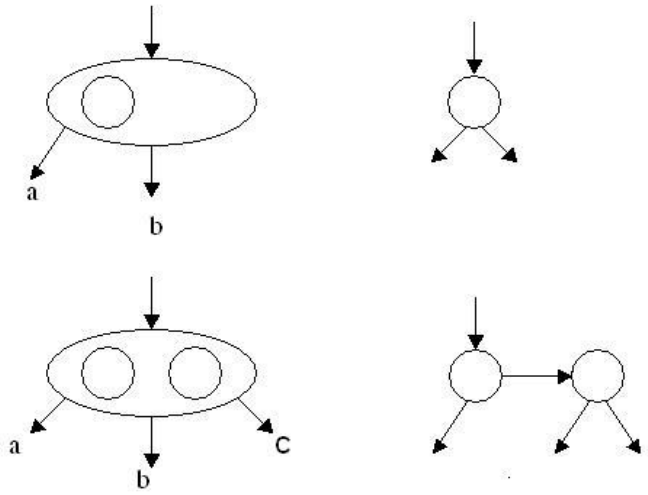
Trong phần trên ta đã tìm hiểu về B-cây cấp n. Với n = 1 ta sẽ có B-cây cấp 1.

Trong trường hợp này thì B-cây cấp 1 không có lợi trong biểu diễn dữ liệu lớn, có thứ tự, có chỉ mục và dùng đến bộ nhớ ngoài; xấp xỉ 50% các trang chỉ chứa một phần tử. Do đó ta xét bài toán tìm kiếm trên cây chỉ bao gồm một mức bộ nhớ (one-level store) - bộ nhớ chính.

Một B-cây nhị phân (BB-cây) bao gồm các nút (trang) mà mỗi trang chứa một hoặc hai phần tử. Như vậy, mỗi trang có hai hoặc ba chỉ điểm chỉ đến các trang con; do đó ta gọi B-cây nhị phân là 2-3 cây (2-3 tree).

Theo định nghĩa của các B-cây thì đối với BB-cây mọi trang lá phải xuất hiện trên cùng một mức và tất cả các trang không phải lá có hai hoặc ba trang con. Vì ta chỉ làm việc với bộ nhớ chính nên ta cần phải sử dụng tối ưu vùng nhớ và việc biểu diễn các phần tử của một nút bằng một mảng là không thích hợp. Một cách giải quyết khác là sử dụng cách cấp phát động, nghĩa là các phần tử của một nút được chứa trong một danh sách liên kết có chiều dài là 1 hoặc 2.

Vì mỗi nút có nhiều nhất ba nút con nên ta cần nhiều nhất ba chỉ điểm mà các chỉ điểm này vừa dùng để chỉ đến các nút con, vừa dùng để chỉ đến phần tử kế tiếp trong danh sách liên kết của nút.



Từ đó, nút B-cây bị mất tính đồng nhất thật sự của nó và các phần tử đóng vai trò của các nút trong một cây nhị phân thông thường. Tuy nhiên ta vẫn phải phân biệt các chỉ điểm chỉ đến các nút con (chiều đứng) với các chỉ điểm chỉ đến phần tử kế tiếp trong cùng một nút (chiều ngang). Chỉ điểm trái bao giờ cũng thẳng đứng và chỉ có các chỉ điểm bên phải là có thể nằm ngang, nên ta dùng một bit (biến h kiểu Boolean) để cho biết chỉ điểm bên phải là nằm ngang hoặc thẳng đứng.

Định nghĩa nút của BB-cây như sau:

```
typedef struct tagNode
{
    int key ;
```

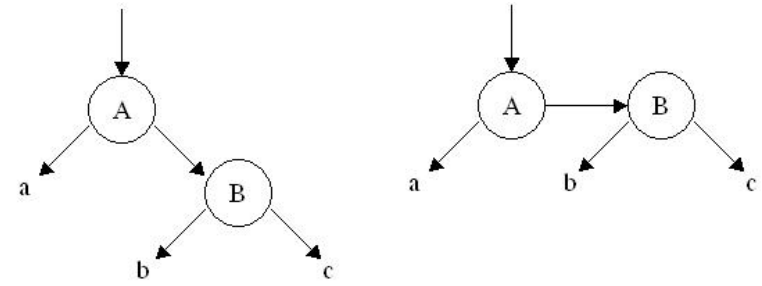
```
    . . .
    struct tagNode* left,*right;
    h  int;
}node;
```

BB-cây được R.Bayer khảo sát vào năm 1971 và biểu diễn một tổ chức cây tìm kiếm đảm bảo chiều dài đường đi cực đại p = 2 * round(logN).

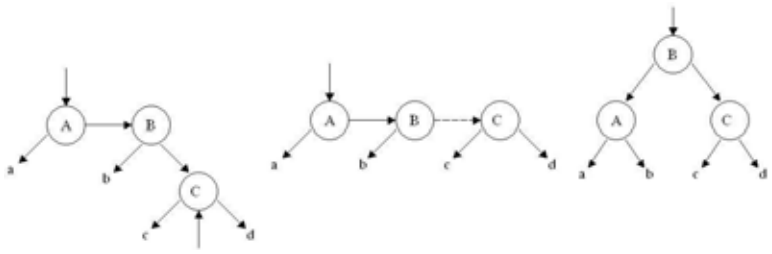
6.2. Các thao tác trên BB-cây

Khi thêm vào BB-cây, ta phải phân biệt bốn trường hợp có thể có từ sự tăng trưởng của các cây con trái hoặc của cây con phải:

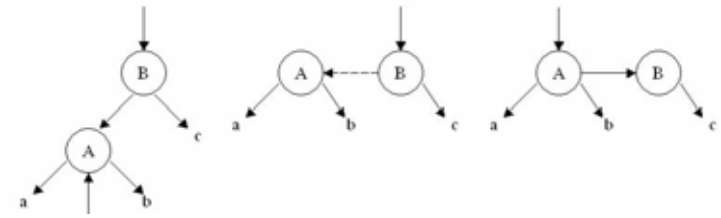
Trường hợp 1: khi cây con bên phải của nút A tăng trưởng và A là khóa duy nhất trên trang này. Khi đó nút con B trở thành phần tử kế tiếp của A trong trang, chỉ điểm phải Right của A nằm ngang.



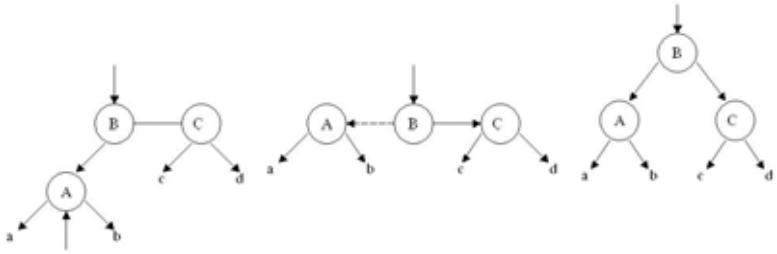
Trường hợp 2: Khi cây con bên phải của nút A tăng trưởng và A có phần tử kế tiếp là B. Khi đó ta có một trang có ba nút nên phải tách trang này thành hai trang và nút giữa B được đưa lên mức trên.



Trường hợp 3: Khi cây con bên trái của nút b tăng trưởng và B là khóa duy nhất trên trang này (chỉ điểm phải Right của B thẳng đứng). Khi đó nút A là phần tử đi trước B trong trang, nhưng chỉ điểm trái Left của B không được nằm ngang nên ta cho chỉ điểm phải Right của A nằm ngang chỉ đến B. Nút gốc là A và chỉ điểm trái Left của B chỉ đến cây con bên trái của A.



Trường hợp 4: Khi cây con bên trái của B tăng trưởng và B có phần tử kế tiếp là C. Khi đó ta có một trang có ba phần tử nên cần phải tách trang này thành hai trang, nút C trở thành nút con của B và nút giữa B được đưa lên mức trên.



Khi tìm kiếm một khóa trên BB-cây thì không có sự khác nhau giữa việc đi theo chỉ điểm nằm ngang hoặc chỉ điểm thẳng đứng. Do đó việc quan tâm đến chỉ điểm trái trở thành nằm ngang trong trường hợp 3 là giả tạo, mặc dù trang của nó vẫn còn chứa chưa quá hai phần tử. Thật vậy, giải thuật thêm vào cây bộc lộ tính không đối xứng lạ thường trong việc xử lý sự tăng trưởng của cây con bên trái và của cây con bên phải, và cho thấy tổ chức của BB-cây tỏ ra giả tạo. Do đó ta cần phải loại bỏ tính không đối xứng của BB-cây.

B-cây nhị phân đối xứng gọi là SBB-cây (symmetric binary B-tree) cũng được R.Bayer khảo sát vào năm 1972. Các giải thuật thêm vào và loại bỏ trên SBB-cây có phức tạp hơn so với BB-cây.

Mỗi nút của SBB-cây cần hai bit (các biến lh và rh có kiểu Boolean) để cho biết bản chất của các chỉ điểm.

Khi thêm vào, ta phải phân biệt bốn trường hợp có thể có từ sự tăng trưởng của các cây con và qua đó cho ta thấy tính thuận lợi của sự đối xứng.

Điều ta quan tâm là giới hạn chiều dài đường đi tối đa là 2*logN, do đó ta chỉ cần đảm bảo là không thể có hai chỉ điểm nằm ngang liên tiếp nhau trên bất kỳ đường tìm kiếm nào. Do đó ta có định nghĩa về SBB-cây như sau:

SBB-cây là một cây có các tính chất sau:

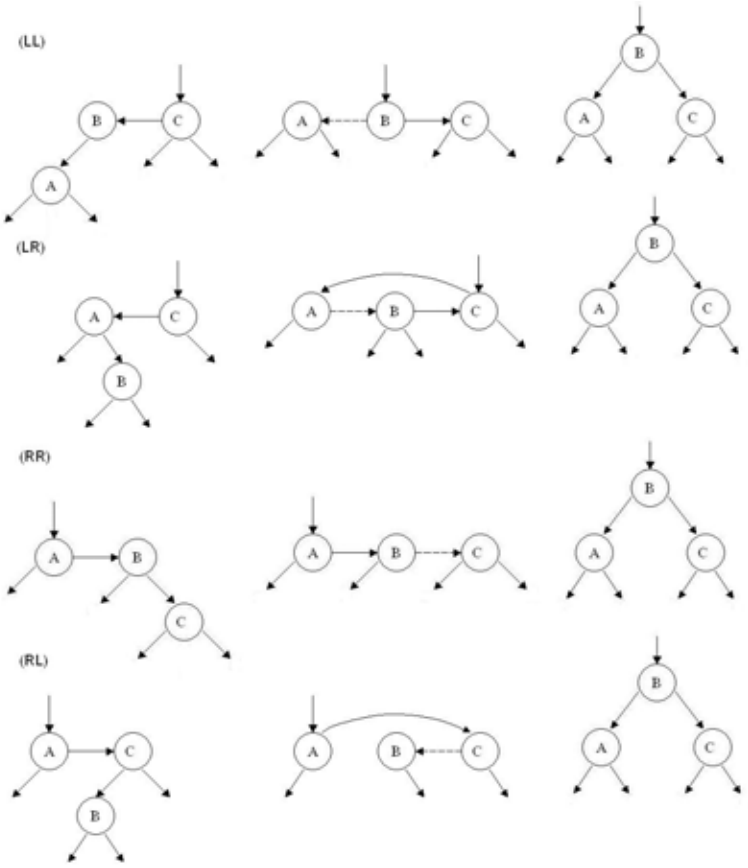
1. Mỗi nút chứa một khóa và có muvh nhất hai cây con (có hai chỉ điểm).

2. Mỗi chỉ điểm hoặc nằm ngang, hoặc thẳng đứng. Không có hai chỉ điểm nằm ngang liên tiếp trên một đường tìm kiếm bất kỳ.

3. Tất cả các nút lá (các nút không có nút con) xuất hiện trên cùng một mức.

Từ định nghĩa này, ta suy ra đường tìm kiếm dài nhất không quá hai lần chiều cao của cây. Vì không có SBB-cây nào có chiều cao lớn hơn round(logN), nên 2 * round(logN), nên 2 * round(logN) là giới hạn trên của chiều dài đường tìm kiếm.

Ví dụ: Xét việc tăng trưởng của SBB-cây khi thêm vào các sau đây với dấu chấm phẩy (;) chỉ ra vị trí đột biến của cây.



Hình 5.61

Các hình này cho ta thấy tính chất thứ ba của các B-cây: tất cả các nút lá đều xuất hiện trên cùng một mức. Người ta có khuynh hướng so sánh các cấu trúc này với các hàng rào vườn vừa được xén bằng. Ta gọi các cấu trúc này là "hàng rào" (hedges). Giải thuật xây dựng các hàng rào-cây (hedge-trees) được trình bày dưới đây. Mỗi nút của cây có hai thành phần lh và rh cho biết các chỉ điểm nằm ngang, có kiểu như sau:

```
1:  typedef struct tagNode
2:  {
3:     int key;
4:     int count;//dem so lan xuat hien cua khoa
5:     struct tagNode* left, *right;
6:     int lh ,rh ;
7:  }node;
```

Thủ tục đệ qui Search dựa trên cơ sở của giải thuật thêm vào cây nhị phân. Tham số thứ ba h cho biết khi nào cây con có gốc là p đã thay đổi và nó tương ứng với tham số h của chương trình tìm kiếm B-cây. Tuy nhiên ta phải chú ý tới ảnh hưởng của việc biểu diễn các trang là các danh sách liên kết: một trang được duyệt bởi một hoặc hai lần gọi thủ tục tìm kiếm. Ta phải

phân biệt giữa trường hợp một cây con (được chỉ ra bởi chỉ điểm thẳng đứng) đã tăng trưởng và một nút anh em (được chỉ ra bởi chỉ điểm nằm ngang) đã nhận thêm một nút anh em khác và từ đó đòi hỏi một sự tách trang. Vấn đề được giải quyết dễ dàng bằng cách cho h lấy một trong ba giá trị sau:

1. h = 0: cây con p không đòi hỏi thay đổi về cấu trúc cây.
2. h = 1: nút p đã nhận thêm một nút anh em.
3. h = 2: cây con p đã tăng chiều cao.

Giải thuật

```
 1: void Search(int x, node *p, int &h)
 2: {
 3:   node* p1,*p2;
 4:   if (p == NULL)//từ chưa có trên cây, thêm từ vào cây
 5:   {
 6:     p = new node;
 7:     h = 2;
 8:     p→key = x;
 9:     count = 1;
10:     left = NULL; right = NULL;
11:     lh = 0; rh = 1;
12:   }
13:   else if (x < p→key)
14:   {
15:     Search(x,p→Left,h);
16:     if (h != 0)
17:         if (p→lh)
18:         {
19:             p1 =p→Left;
20:             h = 2;
21:             p→lh = 0;
22:             if (p1→lh)
23:             {
24:                 p→Left = p1→Right;
25:                 p1→Right = p;
26:                 p1→lh = 0;
27:                 p = p1;
28:             }
29:             else if (p1→rh)
30:             {
```

```
31:                 p2 = p1→Right;
32:                 p1→rh = 0;
33:                 p1→right = p2→Left;
34:                 p2→Left = p1;
35:                 p→Left = p2→Right;
36:                 p2→Right = p;
37:                 p = p2;
38:             }
39:         }
40:     else
41:     {
42:         h = h -1;
43:         if (h != 0)
44:             p→lh = 1;
45:     }
46: }
47: else if (x > p→key)
48: {
49:     Search(x,p→Right,h);
50:     if (h != 0)
51:         if (p→rh)
52:         {
53:             p1 = p→Right;
54:             h = 2;
55:             p→rh = 0;
56:             if (p1→rh)
57:             {
58:                 p→Right = p1→Left;
59:                 p1→Left = p;
60:                 p1→rh = 0;
61:                 p = p1;
62:             }
63:             else if (p1→lh)
64:             {
65:                 p2 =p1→Left;
66:                 p1→lh = 0;
67:                 p1→Left = p2→Right;
68:                 p2→Right = p1;
69:                 p→Right = p1→Left;
70:                 p1→Left = p;
```

```
71:                     p = p2;
72:                 }
73:             }
74:         else
75:         {
76:             h = h -1;
77:             if (h != 0)
78:                 p→rh = 1;
79:         }
80:     }
81:     else
82:     {
83:         p→Count += 1;
84:         h = 0;
85:     }
86: }
```

BÀI TẬP CHƯƠNG 4

Bài tập lý thuyết

1- Hãy trình bày các vấn đề sau đây:
   a- Định nghĩa và đặc điểm của cây nhị phân tìm kiếm.
   b- Thao tác nào thực hiện tốt trong kiểu này.
   c- Hạn chế của kiểu này là gì?

2- Xét thuật giải tạo cây nhị phân tìm kiếm. Nếu thứ tự các khoá nhập vào như sau:

8 3 5 2 20 11 30 9 18 4

thì hình ảnh cây tạo được như thế nào?
Sau đó, nếu hủy lần lượt các nút theo thứ tự sau:

15 20

thì cây sẽ thay đổi như thế nào trong từng bước hủy, vẽ sơ đồ (nêu rõ phương pháp hủy khi nút có cả 2 cây con trái và phải)

3- Áp dụng thuật giải tạo cây nhị phân tìm kiếm cân bằng để tạo cây với thứ tự các khóa nhập vào như sau:

5 7 2 1 3 6 10

thì hình ảnh cây tạo được như thế nào? Giải thích rõ từng tình huống xảy ra khi thêm từng khóa vào cây và vẽ hình minh họa.
Sau đó, nếu hủy lần lượt các nút theo thứ tự như sau:

5 6 7 10

thì cây sẽ thay đổi như thế nào trong từng bước hủy, vẽ sơ đồ và giải thích.

4- Viết các hàm xác định các thông tin của cây nhị phân T

- Số nút lá
- Số nút có đúng 1 cây con
- Số nút có đúng 2 cây con
- Số nút có khoá nhỏ hơn x (giả sử T là CNPTK)
- Số nút có khoá lớn hơn x (giả sử T là CNPTK)
- Số nút có khoá lớn hơn x và nhỏ hơn y (giả sử T là CNPTK)
- Chiều cao của cây
- In ra tất cả các nút ở tầng (mức) thứ k của cây T
- In ra tất cả các nút theo thứ tự từ tầng 0 đến tầng thứ h-1 của cây T (h là chiều cao của cây)
- Kiểm tra xem T có phải là cây cân bằng hoàn toàn không.
- Độ lệch lớn nhất trên cây. (Độ lệch của một nút là độ lệch giữa chiều cao của cây con trái và cây con phải của nó. Độ lệch lớn nhất trên cây là độ lệch của nút có độ lệch lớn nhất).

5- Xây dựng cấu trúc dữ liệu biểu diễn cây N-phân (2<N≤20).

- Viết chương trình con duyệt cây N-phân và tạo sinh cây nhị phân tương ứng với các khoá của cây N-phân.
- Giả sử khoá được lưu trữ chiếm k-byte, mỗi con trỏ chiếm 4 byte, vậy dùng cây nhị phân thay cây N-phân thì có lợi gì trong việc lưu trữ các khoá?.

6- Viết hàm chuyển một cây N-phân thành cây nhị phân.
Giả sử A là một mảng các số thực đã có thứ tự tăng. Hãy viết hàm tạo một cây nhị phân tìm kiếm có chiều cao thấp nhất từ các phần tử của A.

7- Viết chương trình con đảo nhánh (nhánh trái của một nút trên cây trở thành nhánh phải của nút đó và ngược lại) một cây nhị phân.

Bài tập thực hành:

8- Cài đặt chương trình mô phỏng trực quan các thao tác trên cây nhị phân tìm kiếm.

9- Cài đặt chương trình mô phỏng trực quan các thao tác trên cây AVL.

10- Viết chương trình cho phép tạo, tra cứu và sửa chữa từ điển Anh-Việt.

11- Viết chương trình duyệt cây theo mức.

HƯỚNG DẪN

Chương 1

Câu 6:

```
typedef struct
{
        char MNV[9];
        char hoten[31];
        char tinhTrang;
        int soCon;
        char vanHoa[3];
        float luongCB;
}nhanVien;

typedef struct
{
        int coPhep;
        int khongPhep;
        int lamThem;
        char danhGia[3];
        float thucLinh;
}chamCong;
```

Chương 2

Câu 2:

- Trước hết ta đặt giá trị lớn nhất tạm thời bằng số đầu tiên. (Giá trị lớn nhất tạm thời này chính là giá trị lớn nhất ở mỗi giai đoạn của hàm).
- So sánh số kế tiếp trong dãy với giá trị lớn nhất tạm thời và nếu nó lớn hơn giá trị lớn nhất tạm thời thì đặt cho giá trị lớn nhất tạm thời bằng số này.
- Lặp lại bước 2 nếu còn số trong dãy chưa được xét tới.
- Dừng nếu không còn số trong dãy được xét tới. Giá trị lớn nhất tạm thời lúc này chính là giá trị lớn nhất trong dãy số.

Câu 10:

```
//Day con tang co nhieu phan tu nhat
#include <stdio.h>
void main()
{
  int a[10], i, maxstart, maxend, maxlen, tmpstart, tmpend,
        tmplen;
  printf("\nNhap vao 10 phan tu nguyen cua day :");
  for (i=0; i<10; i++)
         scanf("%d", &a[i]);
  printf("Day da cho :\n");
  for (i=0; i<10; i++)
         printf("%6d", a[i]);
```

```
  maxstart = maxend = tmpstart = tmpend = 0;
  maxlen = tmplen = 1;
  for (i=1; i< 10; i++)
  {
        if (a[i] < a[tmpend])
        {
              if (maxlen < tmplen)
              {
                    maxstart = tmpstart;
                    maxend = tmpend;
                    maxlen = tmplen;
              }
              tmpstart = tmpend = i;
              tmplen = 1;
        }
        else
        {
              tmplen++;
              tmpend++;
        }
  }
  if (maxlen < tmplen)
  {
        maxstart = tmpstart;
        maxend = tmpend;
  }
  printf("\nDay tang co so phan tu nhieu nhat la : \n");
  for (i=maxstart; i<=maxend; i++)
       printf("%6d", a[i]);
  getch();
}
```

Câu 13:

```
/* Tron hai mang tang dan thanh 1 mang tang dan */
#include <stdio.h>

#define MAX 10

void main()
{
       int a[MAX], b[MAX], c[2*MAX], n1, n2, i, i1, i2;
       printf("Cho biet so phan tu cua mang thu nhat : ");
       scanf("%d", &n1);
       printf("Nhap vao cac phan tu (tang dan) cua mang thu nhat : ");
       for (i=0; i<n1; i++)
             scanf("%d", &a[i]);
        printf("Cho biet so phan tu cua mang thu hai : ");
        scanf("%d", &n2);
```

```
        printf("Nhap vao cac phan tu (tang dan) cua mang        thu hai : ");
        for (i=0; i<n2; i++)
                scanf("%d", &b[i]);
        i1 = i2 = 0;
        for (i=0; i<n1 + n2; i++)
        {
                if (i1 >= n1 || i2 >= n2)
                        break;
                if (a[i1] < b[i2])
                {
                        c[i] = a[i1];
                        i1++;
                }
                else
                {
                        c[i] = b[i2];
                        i2++;
                }
        }
        if (i1 < n1)
                while (i1 < n1)
                        c[i++] = a[i1++];
        if (i2 < n2)
                while (i2 < n2)
        c[i++] = b[i2++];
        printf("\nCac phan tu cua mang tron : ");
        for (i=0; i<n1+n2; i++)
                printf("%d ", c[i]);
                getch();
}
```

Chương 3

Câu 13:

```
typedef struct  INFO
{
        int HeSo;
        int SoMu;
}INFOTYPE;
typedef struct tagSNODE
{
        INFOTYPE Info;//phan thong tin
        struct tagSNODE*pNext;//con tro den phan tu ke tiep//
}SNODE;
typedef SNODE *PSNODE;//dinh kieu con tro den
//kieu SNODE
typedef struct tagHEAD_SLIST
{
        int nItems;//so phan tu cua danh sach
```

```
	PSNODE pFirst;//con tro den dau danh sach
	PSNODE pLast;//con tro den cuoi danh sach
	PSNODE pCurr;//con tro hien hanh trong
	//danh sach
}HEAD_SLIST;
typedef HEAD_SLIST *PHEAD_SLIST;//dinh kieu
/*Khoi tao mot danh sach rong*/
PHEAD_SLIST	Create(void)
{
	PHEAD_SLIST	pHead;

	pHead=	(PHEAD_SLIST)malloc(sizeof(HEAD_SLIST));
	if (pHead == NULL)
		return NULL;
	pHead->nItems	=	0;
	pHead->pFirst	=	NULL;
	pHead->pLast	=	NULL;
	pHead->pCurr	=	NULL;
	return pHead;
}

/*Huy toan bo danh sach ke ca dau quan ly*/
void	Destroy(PHEAD_SLIST pHead)
{
	PSNODE	pSList;

	pSList=	pHead->pFirst;
	while (pSList)
	{
		free(pSList);
		pSList=	pSList->pNext;

	}
	free(pHead);
}

/*Huy tat ca cac phan tu cua danh sach*/
void	RemoveAll(PHEAD_SLIST pHead)
{
	PSNODE	pSList;

	pSList=	pHead->pFirst;
	while (pSList)
	{
		free(pSList);
		pSList=	pSList->pNext;
	}
	pHead->pFirst	=	NULL;
	pHead->pLast	=	NULL;
	pHead->pCurr	=	NULL;
```

```
	pHead->nItems	=	0;

}
int GetcCount(PHEAD_SLIST pHead)
{
	return pHead->nItems;
}

/*Kiem tra danh sach co rong hay khong*/
int	IsEmpty(PHEAD_SLIST pHead)
{
	return (pHead->nItems == 0) ? 1 : 0;
}

/*	Lay so luong phan tu cua danh sach*/
int	GetCount(PHEAD_SLIST pHead)
{
	return pHead->nItems;
}

/*	Dat con tro hien hanh*/
void	SetCurrent(PHEAD_SLIST pHead, PSNODE pCurr)
{
	pHead->pCurr	=	pCurr;
}

PSNODE	GetCurrent(PHEAD_SLIST pHead)
{
	return pHead->pCurr;
}

/*	lay phan tu dau tien trong danh sach*/
PSNODE	FirstItem(PHEAD_SLIST pHead)
{
	pHead->pCurr	=	pHead->pFirst;
	return pHead->pCurr;
}

/*	lay phan tu ke tiep trong danh sach*/
PSNODE	NextItem(PHEAD_SLIST pHead)
{
	if (pHead->pCurr)
	{
		pHead->pCurr = pHead->pCurr->pNext;
		return pHead->pCurr;
	}
	else
		return NULL;
}
```

```
/*      them phan tu vao cuoi danh sach*/
void    AddTail(PHEAD_SLIST pHead, PSNODE pNew)
{
        if (pHead->pLast == NULL)
        {
                pNew->pNext         =       NULL;
                pHead->pFirst       =       pNew;
                pHead->pLast        =       pNew;
        }
        else
        {
                pNew->pNext         =       NULL;
                pHead->pLast->pNext     =       pNew;
                pHead->pLast        =       pNew;
        }
        pHead->nItems++;
}

/*      them phan tu vao dau danh sach*/
void    AddHead(PHEAD_SLIST pHead, PSNODE pNew)
{
        if (pHead->pFirst == NULL)
        {
                pNew->pNext         =       NULL;
                pHead->pFirst       =       pNew;
                pHead->pLast        =       pNew;
        }
        else
        {
                pNew->pNext         =       pHead->pFirst;
                pHead->pFirst       =       pNew;
        }
        pHead->nItems++;
}

/*chen phan tu vao truoc phan tu moc cua danh sach */
void    InsertBefore(PHEAD_SLIST pHead, PSNODE pHospot, PSNODE pNew)
{
        if (pHead->pFirst == pHospot)
                AddHead(pHead, pNew);
        else
        {
                PSNODE      pPrev, pCurr;
                pCurr =     pHead->pFirst;
                while (pCurr)
                {
                        pPrev =     pCurr;
                        pCurr =     pCurr->pNext;
                        if (pCurr == pHospot)
                                break;
```

```
		}
		if (pCurr != NULL)
		{
			pPrev->pNext	=	pNew;
			pNew->pNext	=	pCurr;
		}
	}
}

/*chen phan tu vao sau phan tu moc cua danh sach*/
void	InsertAfter(PHEAD_SLIST pHead, PSNODE pHospot, PSNODE pNew)
{
	if (pHead->pLast == pHospot)
		AddTail(pHead, pNew);
	else
	{
		PSNODE	pCurr;
		pCurr =	pHead->pFirst;
		while (pCurr)
		{
			if (pCurr == pHospot)
				break;
			pCurr =	pCurr->pNext;
		}
		if (pCurr != NULL)
		{
			pNew->pNext = pCurr->pNext;
			pCurr->pNext = pNew;
		}
	}
}

/*huy phan tu moc cua danh sach*/
void	RemoveAt(PHEAD_SLIST pHead, PSNODE pHospot)
{
	if (IsEmpty(pHead))
		return;
	if (pHospot == pHead->pFirst)
	{
		pHead->pFirst = pHead->pFirst->pNext;
		free(pHospot);
	}
	else
	{
		PSNODE	pPrev, pCurr;
		pCurr =	pHead->pFirst;
		while (pCurr)
		{
			pPrev =	pCurr;
			pCurr =	pCurr->pNext;
```

```
                if (pCurr == pHospot)
                        break;
                }
                if (pPrev != pHead->pLast)
                {
                        pPrev->pNext = pCurr->pNext;
                        if (pCurr == pHead->pLast)
                                pHead->pLast = pPrev;
                        free(pCurr);
                }
        }
        pHead->nItems--;
}
void hoanvi(INFO &P,INFO &Q)
{
        INFO T;
        T=P;
        P=Q;
        Q=T;
}
//Sort List
void ListStraiSelection(PHEAD_SLIST  pHead)
{
        PSNODE P,Q,Min;
        P=pHead->pFirst;
        while(P)
        {
                Q=P->pNext;  Min=P;
                while(Q)
                {
                        if(Q->Info.SoMu<Min->Info.SoMu)
                                Min=Q;
                        Q=Q->pNext;
                }
                hoanvi(Min->Info,P->Info);
                P=P->pNext;
        }
 }

void Traverse(PHEAD_SLIST pHead)
{
        PSNODE pSList;
        pSList=        FirstItem(pHead);
        textcolor(YELLOW);
        if(pSList)
        {
                while (pSList->pNext)
                {
                        printf ("%dX^%d+",pSList->Info.HeSo,pSList->Info.SoMu);
                pSList = NextItem(pHead);
```

```
                }
                printf("%dX^%d",pSList->Info.HeSo,pSList->Info.SoMu);
        }
}
PSNODE Search(PHEAD_SLIST pHead, PSNODE pNew)
{
        PSNODE      pSList, pR;
        int Found=0;
        pSList=       pHead->pFirst;
        while ((pSList) && (!Found))
        if((pSList->Info.HeSo==pNew->Info.HeSo)&&(pSList->Info.SoMu==pNew-
        >Info.SoMu))
                Found=1;
        else
                pSList=       pSList->pNext;
        return pSList;
}
PSNODE SearchSoMu(PHEAD_SLIST pHead, PSNODE pNew)
{
        PSNODE      pSList;
        int Found=0;
        pSList=       pHead->pFirst;
        while ((pSList) && (!Found))
        if(pSList->Info.SoMu==pNew->Info.SoMu)
                Found=1;
        else
                pSList=       pSList->pNext;
        return pSList;
}
void InsertDT(PHEAD_SLIST pHead, PSNODE pNew)
{
         PSNODE pCurrent,pSList;
         pSList         =FirstItem(pHead);
         while ((pSList)&&(pSList->Info.SoMu > pNew->Info.SoMu))
         {
                pCurrent       =       pSList;
                pSList=       NextItem(pHead);

         }
        if(pSList->Info.SoMu==pNew->Info.SoMu)
                pSList->Info.HeSo+=pNew->Info.HeSo;
         else
        {
                pNew->pNext=pSList;
                if(pSList==FirstItem(pHead))
                        pHead->pFirst=pNew;
                else
                        pCurrent->pNext=pNew;
        }
}
```

```
void   main(void)
{
        randomize();
        PHEAD_SLIST  pHead1   ,         pHead2;
        PSNODE      pSList1,pSList2,;
        PSNODE      pR,P,pN,P1,PNew,pCurrent;
        long    i,x;
        int ch,SM,HS;
        clrscr();
         pHead1        =        Create();
         pHead2        =        Create();
        do
        {
                textattr(26);
                printf("\n1.Nhap DaThuc \n");
                printf("2.Cong DaThuc\n");
                printf("3.Nhan DaThuc\n");
                printf("4.Chia DaThuc\n");
                printf("5.DeleteX DaThuc\n");
                printf("6.SearchX DaThuc\n");
                printf("7.Sort DaThuc\n");
                printf("8.Traverse DaThuc\n");
                printf("9.Sum Node DaThuc\n");
                printf("10.Dao Ham DaThuc\n");
                printf("0.Exit\n");
                switch(ch)
                {
                        case 1:{
                        printf("\nNhap Dathuc 1:\n");
                        pSList1=(PSNODE)malloc(sizeof(SNODE));
                        if (pSList1 == NULL)
                                break;
                        printf("\nNhap HS: ");
                        scanf("%d",&HS);
                        pSList1->Info.HeSo=HS;
                        printf("\Nhap SM: ");
                        scanf("%d",&SM);
                        pSList1->Info.SoMu=SM;
                        InsertDT(pHead1,pSList1);
                        Traverse(pHead1);
                        printf("\nNhap Dathuc 2:");
                        pSList2         =(PSNODE)malloc(sizeof(SNODE));
                        if (pSList2 == NULL)
                                break;
                        printf("\nNhap HS: ");
                        scanf("%d",&HS);
                        pSList2->Info.HeSo=HS;
                        printf("\Nhap SM: ");
                        scanf("%d",&SM);
```

```
pSList2->Info.SoMu=SM;
InsertDT(pHead2,pSList2);
Traverse(pHead2);
}break;
case 2:{
pN=FirstItem(pHead2);
while(pN)
{
        InsertDT(pHead1,pN);
        pN = NextItem(pHead2);
 }
 textcolor(RED);
 cprintf("\nTong 2 Da Thuc: ");
Traverse(pHead1);
}break;
case 3:{
}break;
case 4:{
}break;
case 5:{
printf("\nNhap HS: ");
scanf("%d",&HS);
PNew->Info.HeSo=HS;
printf("\Nhap SM: ");
scanf("%d",&SM);
PNew->Info.SoMu=SM;
pR=Search(pHead1,PNew);
RemoveAt(pHead1,pR);
}break;
case 6:{
}break;
case 7:{
Traverse(pHead1);
printf("\n Da Thuc 1 sau khi sort\n");
ListStraiSelection(pHead1);
Traverse(pHead1);
Traverse(pHead2);
printf("\nDa Thuc 2 sau khi sort\n");
ListStraiSelection(pHead2);
Traverse(pHead2);
}break;
case 8:{
printf("\nDa thuc 1: \n");
Traverse(pHead1);
printf("\nDa thuc 2: \n");
Traverse(pHead2);
}break;
case 9:
printf("\nSum Node List:%4d ",GetCount(pHead1));
break;
```

```
                case 10: break;
            }
            printf("\nChoice(1.2.3.4.5.6.7.8.9.10.0): ");
            scanf("%d",&ch);
            clrscr();
        }while(ch);
}
```

Chương 04

Câu 4: Chương trình đếm số lá của cây

```
typedef int element_type;
typedef struct node {
        element_type element;
        struct node *left, *right;
} NODE;

NODE *root;

void khoi_tao_cay(NODE ** root)
{
        *root = NULL;
}

void insert(NODE *tmp, NODE **root)
{

        if (tmp->element < (*root)->element)
                if ((*root)->left)
                        insert(tmp, &(*root)->left);
                else
                        (*root)->left = tmp;
        else
                if ((*root)->right)
                        insert(tmp, &(*root)->right);
                else
                        (*root)->right = tmp;
}

void insert_node(element_type e, NODE **root)
{
        NODE *tmp;

        tmp = (NODE *)malloc(sizeof(NODE));
        tmp->element = e;
        tmp->left = NULL;
        tmp->right = NULL;
        if (*root == NULL)
                *root = tmp;
```

```
        else
                insert(tmp, root);
}

void nhap_cay(NODE **root)
{
        element_type e;
        do {
                printf("\nNhap element (-1 de ket thuc) : ");
                scanf("%d", &e);
                if (e != -1)
                        insert_node(e, root);
        } while (e != -1);
}

int dem_nut_la(NODE *root)
{
        if (root == NULL)
                return 0;
        else
                if (root->left != NULL || root->right != NULL)
                    return dem_nut_la(root->left) + dem_nut_la(root->right);
                else
                    return 1;
}

void main()
{
        int tong_nut_la;
        khoi_tao_cay(&root);
        nhap_cay(&root);
        tong_nut_la = dem_nut_la(root);
        printf("\nTong so nut la     = %d", tong_nut_la);
        getch();
}
```

Câu 11:

```
#define MAX 100

typedef int element_type;
typedef struct node {
  element_type element;
  struct node *left, *right;
} NODE;

NODE *queue[MAX + 1];
int front, rear, queue_size;

void khoi_tao_queue()
```

```
{
  front = rear = 0;
  queue_size = 0;
}

int is_empty()
{
  return (queue_size == 0);
}

int is_full()
{
  return (queue_size == MAX);
}

int push(NODE *value)
{
  if (queue_size < MAX)
  {
    queue_size++;
    queue[rear++] = value;
    if (rear == MAX)
      rear = 0;
  }
  return rear;
}

int pop(NODE **value)
{
  if (queue_size > 0)
  {
    *value = queue[front++];
    if (front > MAX)
      front = 0;
    queue_size--;
  }
  return front;
}

NODE *root;

void khoi_tao_cay(NODE ** root)
{
  *root = NULL;
}

void insert(NODE *tmp, NODE **root)
{

  if (tmp->element < (*root)->element)
```

```
    if ((*root)->left)
      insert(tmp, &(*root)->left);
    else
       (*root)->left = tmp;
  else
    if ((*root)->right)
      insert(tmp, &(*root)->right);
    else
       (*root)->right = tmp;
}

void insert_node(element_type e, NODE **root)
{
  NODE *tmp;

  tmp = (NODE *)malloc(sizeof(NODE));
  tmp->element = e;
  tmp->left = NULL;
  tmp->right = NULL;
  if (*root == NULL)
    *root = tmp;
  else
    insert(tmp, root);
}

void nhap_cay(NODE **root)
{
 element_type e;
 do {
   printf("\nNhap element (-1 de ket thuc) : ");
   scanf("%d", &e);
   if (e != -1)
     insert_node(e, root);
 } while (e != -1);
}

void duyet_cay_level(NODE *root)
{
 NODE *p;
 khoi_tao_queue();
 if (root != NULL)
   push(root);
 while (!is_empty())
 {
   pop(&p);
   printf("%d ", p->element);
   if (p->left != NULL)
     push(p->left);
   if (p->right != NULL)
     push(p->right);
```

```
  }
}

void main()
{
  khoi_tao_cay(&root);
  nhap_cay(&root);
  duyet_cay_level(root);
  getch();
}
```

TÀI LIỆU THAM KHẢO

1- Trần Hạnh Nhi – Dương Anh Đức : **Giáo trình Cấu trúc dữ liệu và Giải thuật** – Đại học quốc gia thành phố Hồ Chí Minh – 2001.

2- Nguyễn Trung Trực : **Cấu trúc dữ liệu** – Trường Đại học Bách khoa thành phố Hồ Chí Minh – 1994

3- Niklaus Wirth : **Algorithms + Data Structures = Programs** – Prentice Hall – 1976

4- Robert Sedgewick : **Cẩm nang thuật toán** – Nhà xuất bản Khoa học Kỹ thuật – 1994

5- Th. Cormen, Ch. E. Leiserson, R. L. Rivest, **Introduction to Algorithms** (MIT Press, 1998).

6- A.V. Aho, J.E Hopcroft, J.D Ulman, **Data structures and algorithms** (Addison Wesley, 1983).

Mục lục